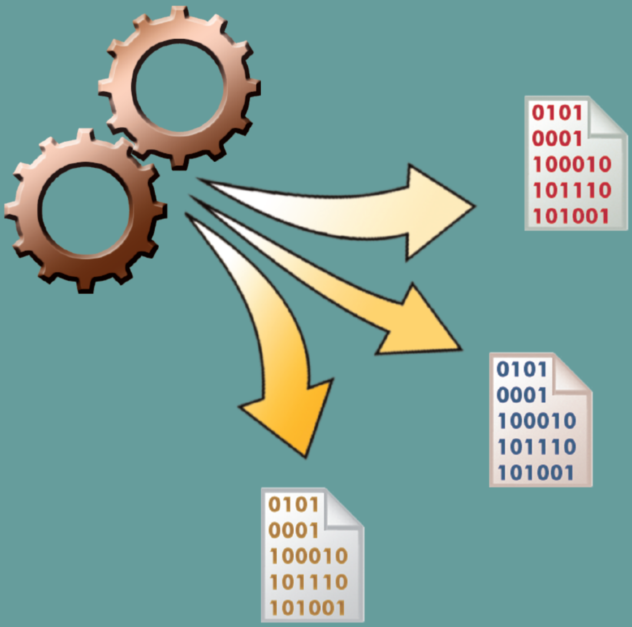
Victor Pankratius

Product Lines for Digital Information Products



universitätsverlag karlsruhe

Victor Pankratius

Product Lines for Digital Information Products

Product Lines for Digital Information Products

by
Victor Pankratius

universitätsverlag karlsruhe

Dissertation,
genehmigt von der Fakultät für Wirtschaftswissenschaften
der Universität Fridericiana zu Karlsruhe, 2007

Referent:	Prof. Dr. Wolffried Stucky
Korreferenten:	Prof. Dr. Christof Weinhardt
	Prof. Dr. Dieter Rombach

Impressum

Universitätsverlag Karlsruhe
c/o Universitätsbibliothek
Straße am Forum 2
D-76131 Karlsruhe
www.uvka.de

Cover design: Victor Pankratius

Universitätsverlag Karlsruhe 2007
Print on Demand

ISBN: 978-3-86644-133-0

To my wife and my parents.

Preface

Motivated by the ongoing trend of digitization which has fundamental implications on how we create, use, and deliver information products, this thesis shows how variants of digital information products can be engineered. Many people have supported me in various ways to finish this thesis while I was a research assistant at the Institute of Applied Informatics and Formal Description Methods at the University of Karlsruhe. I would like to thank them all for their support.

In particular, I would like to thank my advisor, dean of the faculty and head of the institute, Prof. Dr. Wolffried Stucky, for his continuous support, for the fruitful collaboration, and for the great opportunities and liberties he offered me throughout my research. I also thank my secondary advisors, Prof. Dr. Christof Weinhardt and Prof. Dr. Dieter Rombach, for the helpful discussions, and the other members of the Ph.D. committee, Prof. Dr. Andreas Oberweis and Prof. Dr. Hagen Lindstädt, for the time spent reviewing my thesis. I also thank Prof. Dr. Gottfried Vossen for his constructive comments and general advice during the early stages of my Ph.D. studies.

Furthermore, I thank my colleagues in our Business Information and Communication Systems group for the cooperation and the pleasant working atmosphere, as well as the student assistants and the students of the Software Engineering Tools lab who contributed with great enthusiasm to the tool developed for this thesis.

I warmly thank my wife and my parents for their continuous love, support, and patience.

Finally, I would like to point the readers to the associated Web site where this book and supplementary material can be downloaded:

http://www.product-lines-for-digital-information-products.com

Karlsruhe, February 2007 *Victor Pankratius*

Summary

The growth of the Web has spurred the creation and exchange of products which exist in digital form only, and which are created, distributed, and used exclusively in digital form. Digital information products are an important class of widely used digital products, whose core benefit lies in the delivery of information or education (e.g., electronic books, online newspapers, e-learning courses).

Variants of digital information products are often created to satisfy different customer needs (e.g., a light/standard/full version). Currently, such variants are often created in an unsystematic way, and there are hardly any attempts to systematize the creation of digital information products in this context. This is especially problematic after their initial creation when updates have to be made and the variability of possible changes is not planned and limited in advance. In addition, potential commonalities between such variants are either not exploited at all or not systematically enough, which typically leads to redundancy and inconsistencies during creation or updates.

This thesis introduces a novel and systematic approach, called Product Lines for Digital Information Products (PLANT), which is a special approach focusing on the creation of variants of digital information products within a product line, and which tackles the aforementioned problems in an engineering-like way by adapting and extending concepts from the area of software product lines. Based on a systematic process model consisting of family engineering, domain engineering, and application engineering, PLANT allows common product requirements to be defined, commonalities and differences of content in product variants to be planned and limited in advance using a conceptual model, and variants to be created according to a construction workflow model that can draw upon prepared components. An associated tool is introduced as well, which handles the product configurations and the construction workflow model in a novel and integrated way. The feasibility and advantages of PLANT are finally demonstrated in a case study related to e-learning.

Contents

Part I Fundamentals

Part II Product Lines for Digital Information Products

Part III Tool Support & Case Study

1

Introduction

Many parts of today's economy are reshaped by an increasing digitization which is spurred not only by the ongoing development of hardware and software, but also by the diffusion of technology into everyday life. Especially the growth of the Web has facilitated the creation and distribution of new kinds of products that are available and distributed in an entirely digital form only (i.e., their form is not "material" in the traditional sense), like for example, software, digital music, online publications, online games, or online video, to name just a few.

An important category of digital products are the digital information products whose core benefit lies in the delivery of information or education. Digital information products are ubiquitous and can be found in various areas, like for example: electronic books, electronic user and training manuals, electronic newspapers, electronic learning content. The content, which actually contains information, plays an important role for these products and is one of the main aspects contributing to a competitive advantage. It is therefore of utmost importance for the business success of producers to create such products in an efficient way and adapt them to the individual needs of particular customers or market segments. To make the production of digital information products more efficient, reuse of content is needed. However, numerous problems are currently associated with reuse in this context, especially from a technical and organizational point of view, as presented next. Thereafter, the goals and approaches of this thesis are introduced, which are aimed to tackle the presented problems, and the structure of the thesis is outlined.

1.1 Current Problems

In the early days of the Web, the amount of information contained in digital products was small enough to be handled manually. Meanwhile, over 90% of the currently produced information are estimated to be in digital format, while the information available in other physical formats is estimated to become in most parts digitized [Var05]. These changed circumstances have an impact on several aspects of the creation of digital information products.

In many cases, digital information products are versioned, i.e., a base product is created with different levels of quality or slightly differing content or features [Var97]. This way, digital information products can be tailored to the needs of different customer groups, and saturation effects in markets – which can occur due to the indestructibility of digital products in general – can be compensated (see Chapter 2 for details). The key to the technical realization of versioning is the modularization of content, so that different modules or content components can be reused within different configurations of versions of information products.

Although modularization and reuse of content may seem easy to realize, there are currently several technical and organizational problems. The reuse of content components is often not planned in advance, but done adhoc. Therefore, components found in repositories often have different underlying assumptions with respect to file types, layout, structure, or presentation. Reusing such components for the creation of information products may require modifications which can be just as tedious as a re-creation from scratch. The chosen component models, which define a logical structure for a component, may diverge as well. In addition, there are many different standards, especially for metadata (see Chapter 3), which also differ depending on the chosen application area. Especially when several concurrent standards exist, exchange and reuse of content is complicated. Another neglected aspect is that content is often not reused in isolation, but in a certain reuse context, for example with some other related content; such a reuse context is typically not planned in advance. In many cases, too much individualization makes reuse more difficult, and only certain variants of an information product are really needed, in which for example only a few predefined content configurations of a base product are offered. In such scenarios, different versions of information products may have commonalities that should be modeled in an explicit way in order to simplify updates for common parts.

The presented problems are discussed in more detail for the example area of e-learning in Chapter 3. Examples from e-learning are also offered throughout the thesis, as the concentration on a certain area allows the presentation of more subtle details than a more general treatment.

1.2 Goals & Approaches of the Thesis

From the aforementioned problems, it becomes obvious that the availability of standards is not a sufficient solution for reuse. A structured, engineering-like approach is needed to tackle the reuse problem for digital content from a technical as well as organizational point of view.

The thesis introduces such an approach, called Product Lines for Digital Information Products (PLANT), that transfers and adapts concepts from the area of software product lines (Chapter 4) to that of digital information products (described in Chapters 2 and 3). The PLANT approach is an integrated "package" that is able to handle technical and organizational aspects, and consists of a strategy, a process model, and an associated tool. The thesis is interdisciplinary and draws upon insights from a variety of fields, such as software engineering, information systems, workflow management, databases, economics, management, and e-learning.

The reason to adapt and extend software product lines to this context is that they represent a solution to similar reuse problems that were experienced for software components in software engineering. Software product lines focus on reuse within sets of related programs by first evaluating possible commonalities and variabilities (i.e., differences), and creating a set of parameterizable core assets that are exclusively used thereafter to create software products in a predominantly generative manner. The main differences to other approaches (e.g. [HSI05]) are that product lines proactively prepare reuse and define the reuse context from the beginning, and impose restrictions on the allowed configurations (i.e., realizable product variants) in the reuse context. This procedure in fact establishes an additional layer of standardization, whose scope is only the set of products that belong to one product line. Additional details are discussed in Chapter 4.

Similar to software product lines, PLANT focuses on the creation of similar digital information products that are part of one product line, which can be created in a generative manner using predefined core assets (e.g., content components, layout specifications). For example, an e-learning course in Information Systems can have a different configuration of topics, depending on the targeted audience (e.g., beginners/experts or database-centric/workflow-centric), and the content for these topics is prepared in advance, while a specific course is generated only upon request. However, PLANT also defines and restricts in advance the possible content configurations for information products, and imposes uniform specifications for all content components. Although this may sound negative at first, it actually simplifies and improves reuse in several respects:

- the reuse context is made explicit, and content developers now know from the beginning in which setting the developed context will be used;
- since all content components have the same underlying assumptions, a developer can be sure that a composition of core assets belonging to one

product line will finally work, without encountering the aforementioned problems; this is also true for public repositories.

- an explicit design of parts that are common to several information products makes updates easier and reduces potential inconsistencies.
- product lines support standardization as well as individualization for information products.

PLANT makes a tradeoff and tries to find a balance between individualization and standardization of information products, with the aim that a content composition approach becomes feasible under realistic circumstances, and reuse more efficient. PLANT is introduced in Chapter 5, and other details on it in the Chapters 6–10.

1.3 Outline of the Thesis

The thesis is structured into three parts (see Fig. 1.1).

- **Part I** introduces in the Chapters 2–4 the fundamentals needed for part II and part III.
- **Part II** presents in the Chapters 5–8 the concepts of the Product Lines for Digital Information Products (PLANT) Approach.
- **Part III** shows in Chapters 9–10 available tool support to develop information products with PLANT, and a case study with a concrete case where PLANT has been applied to show the feasibility of the approach.

In particular, the chapters discuss the following topics:

- **Chapter 2:** discusses the notion of a product, digital product, and digital information product. The typical characteristics and classifications of digital products are used to point out the differences to traditional products, and why versioning is needed for digital products. Special characteristics are also discussed in a value chain context.

- **Chapter 3:** focuses on e-learning as an example area where digital information products occur in a form of so-called learning objects. For this area, the typical infrastructure and systems architecture are outlined. This area is used as a concrete example to show existing problems with content reuse for information products.

- **Chapter 4:** introduces the software product line concept, which is a special reuse concept for the creation of variants of software products.

- **Chapter 5:** introduces the Product Lines for Digital Information Products (PLANT) Approach, which transfers and adapts the key concepts from software product lines (Chapter 4) to the context of digital information products (Chapter 2 and 3). The development process in PLANT

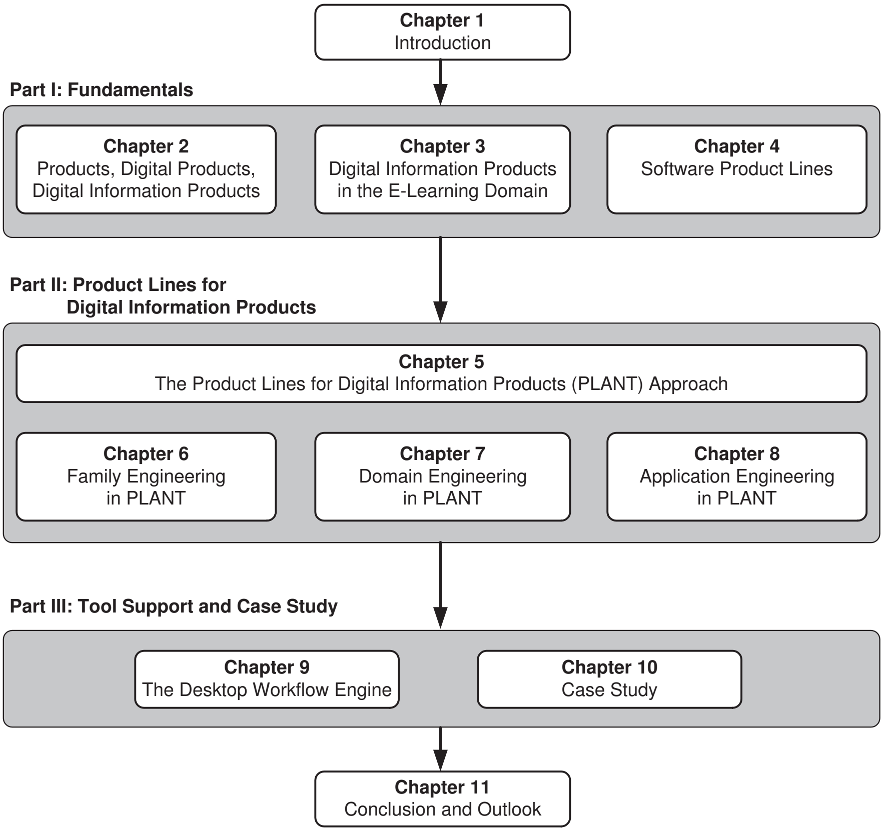

Chapter 1
Introduction
Part I: Fundamentals
Chapter 2
Products, Digital Products, Digital Information Products
Chapter 3
Digital Information Products in the E-Learning Domain
Chapter 4
Software Product Lines
Part II: Product Lines for Digital Information Products
Chapter 5
The Product Lines for Digital Information Products (PLANT) Approach
Chapter 6
Family Engineering in PLANT
Chapter 7
Domain Engineering in PLANT
Chapter 8
Application Engineering in PLANT
Part III: Tool Support and Case Study
Chapter 9
The Desktop Workflow Engine
Chapter 10
Case Study
Chapter 11
Conclusion and Outlook

Fig. 1.1. Structure of the thesis.

is divided into the sub-processes family engineering, domain engineering, and application engineering.

- **Chapter 6**: presents the details of the family engineering sub-process which focuses on the product line as such and deals with management and organizational issues.

- **Chapter 7**: presents the details of the domain engineering sub-process which focuses on all information products in one product line and on preparations for the reuse of content.

- **Chapter 8**: presents the details of the application engineering sub-process which focuses on the generation of a single information product, based on the preparations made in domain engineering.

- **Chapter 9**: describes the Desktop Workflow Engine which is a tool that supports the creation of digital information products in domain engineering and application engineering.

- **Chapter 10**: shows a case study where PLANT has been applied in an e-learning context, to show the feasibility in a concrete application context.

- **Chapter 11**: summarizes the main insights and shows potential extensions and other development directions for the PLANT Approach.

Part I

Fundamentals

2

Products, Digital Products, Digital Information Products

This Chapter introduces and defines the basic notions of products, digital products, and digital information products, which will be used throughout this work. Firstly, the notion of a product is surveyed in the literature and relevant product classifications are presented. The results are then contrasted with the notions of digital products in the literature, and the main aspects are distilled in a working definition for digital products. As a further refinement, digital information products are identified as a special type of digital products. The typical characteristics of digital products – which are different from physical products – are worked out to demonstrate that they influence the economic viability of digital products in a competitive marketplace. A classification scheme for digital products describes relevant criteria for a comparison of digital products, and helps derive strategic decisions. Thereafter, a value chain model for digital information products is introduced, which offers a general perspective on relevant activity areas. Finally, based on the insights gained in this Chapter, some critical points are identified which motivate the approach taken in the Chapters to follow.

2.1 Products in General

Although the word “product” is frequently used in everyday life, the exact notion of a product is fairly complex and difficult to delimit, especially with respect to physical and non-physical properties. There are several approaches to define products, which are presented next. Thereafter, some widely known classifications are introduced.

2.1.1 Definitions

As a starting point of the analysis of existing product definitions, dictionary entries for the word “product” point to its latin origin. It is derived from the latin word “producere” which means “to bring out” [Bro92, Uni07]. The

Oxford Dictionary of International Business characterizes a product as "thing or substance produced, esp. by manufacture" [Oxf98], and [Gab04] defines a product as the result of the production and the aims of a company, as well as a means to satisfy needs. Products are understood as outputs or results of a production process which combines or transforms inputs or production factors (for details, esp. on production factors, see [WD05]). According to the German Product Liability Law, every movable thing can be a product, even if it is a part of another movable thing. This purely physical understanding is extended by explicitly defining electricity as a product [Pro06].

In the marketing literature, the product notion defined by Kotler and Armstrong has gained substantial influence and considers products as [KA05]: "anything that can be offered to a market for attention, acquisition, use, or consumption that might satisfy a want or need. Products include more than just tangible goods. Broadly defined, products include physical objects, services, events, persons, places, organizations, ideas, or mixes of these entities". The authors intend to broaden the concept of a product. Persons (e.g., politicians, entertainers, sports teams) are seen as products in a sense that activities can be undertaken to create, maintain, or change public attitudes or behavior towards them. During image advertising campaigns, organizations can be presented like products. Similarly, places or touristic regions may compete to attract tourists.

On a closer examination, Kotler and Armstrong's product notion distinguishes between three conceptual levels, as depicted in Fig. 2.1. At the center, it is assumed that each product has some form of *core benefit* which provides problem-solving capabilities. The second level turns the core benefit into the *actual product*: *product quality* determines the degree at which a product is able to perform its functions; *features* are used to distinguish between various models of a product, as well as to distinguish the product from the competition; *design* contributes to the looks of a product, but also to its usefulness; a *brand*, which can be a name, term, sign, symbol, or a combination thereof, can add value by providing a means for identification, quality expectation and differentiation; the *package* acts as a container or wrapper for the product. At the third level, the *augmented product*, additional value is generated by services and benefits related to *delivery and credit, installation, warranty*, or *after-sale service*. Kotler and Armstrong regard services as a special form of products [KA05], which consist of "activities, benefits, or satisfactions offered for sale that are essentially intangible and do not result in the ownership of anything". However, services have some special characteristics, like *intangibility* (i.e., they cannot be felt, heard, tasted, or seen before purchase), *inseparability* (they cannot be separated from the providers), *perishability* (they cannot be stored for later use), and *variability* (their quality depends on who provides them when, where, and how).

The product model assumes that each of the conceptual levels contributes additional value to the product. From a practical point of view, the value in each layer of a product is created in the value chain of a company [Por80],

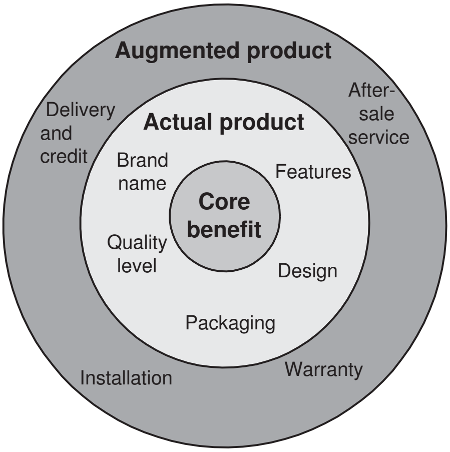


Fig. 2.1. Product model with three conceptual levels [KA05].

which consists of activities and processes that contribute to the profit margin and add value for the customer. The value chain of a company is also linked to the value chains of its suppliers and customers. Moreover, it has to be critically remarked that the characteristics presented in the three layers seem more closely related to physical products and do not always match the broad product definition without adaptation. For example, "features" may describe the features of a physical product, but for a person they should be understood as the abilities which characterize the person. Another critical point is that, for a given product, it can be difficult to clearly demarcate the boundaries between the three levels.

A similar product model has been developed by Vershofen [Ver40]. The model describes a product as having a *basic benefit* and an *additional benefit*. The basic benefit is derived from its physical, chemical, or technical properties and has problem-solving capabilities, similar to the core benefit of Kotler and Armstrong. The additional benefit of a product is assumed to be on a "mental" level. It is influenced by an esthetic design, the package, or the brand prestige. For example, a very expensive pen has the basic benefit that one can take notes, but the extended benefit that one can demonstrate the membership in a particular group. However, the distinction of basic and additional benefit in this approach is subjective and highly dependent on the view of a particular customer.

Brokhoff touches some additional aspects in his product definition [Bro99]. From his point of view, a product consists of a bundled, indivisible set of properties that need not be restricted to physical artifacts (i.e., a property can be immaterial). The bundling of this set of properties is tailored to a prospective (known or unknown) user and is intended to be exchanged, and through the return of the exchange, fulfill the goals of the supplier. For example, the features of a particular car (engine and seat type, color), as well as other benefits like those delivered by services (free repair during warranty period), constitute the bundle characterizing the product “car”. However, in some cases it might be difficult to exactly define the properties which constitute a product. The view that products have material and immaterial parts which implement a product’s functions is also shared by [Gab04].

In the following, the notion of a product will be used in a sense that incorporates the main aspects presented previously. Building upon the definition by Brokhoff, a product will be understood as a bundle of tangible or intangible properties, features, or services, which is offered by a supplier with the intention to fulfil his goals. Furthermore, the bundle can be tailored to a user. Each of the product properties is assumed to belong to one of the three layers of core benefit, actual product, and augmented product, as understood by Kotler and Armstrong. It is not contradictory to assume that properties contribute to the basic product benefit or to the additional benefit as proposed by Vershofen.

2.1.2 Classifications

In the literature, a considerable number of classifications for products has been developed. Therefore, only the ones considered most important for the following examinations are discussed (cf.Fig. 2.2). For other classifications, the reader is referred to [Zan02, Gab04, WD05].

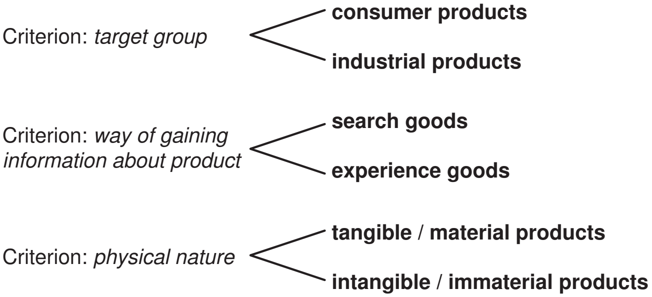

Criterion: *target group*
consumer products
industrial products
Criterion: *way of gaining information about product*
search goods
experience goods
Criterion: *physical nature*
tangible / material products
intangible / immaterial products

Fig. 2.2. Some important product classification schemes

In general, based on the target group who uses the products, two broad classes are distinguished [KA05]: *consumer products* and *industrial products.* Consumer products are bought by final customers for personal consumption, whereas industrial products are bought by individuals or organizations for business purposes.

The classification introduced by Nelson is based on how consumers gain information about products [Nel70]. It distinguishes between *search goods* and *experience goods*. Search goods are products whose quality can be assessed without experiencing them. Usually, a visual inspection of the product or its characteristic information suffices to asses all relevant properties prior to purchase (e.g., a TV set). By contrast, the perceived value of experience goods builds upon the actual experience and the accumulated knowledge using the product (e.g., a dinner in a restaurant). Therefore, branding, previewing, customer reviews, or reputation play an important role for experience goods, and may help transform experience goods into search goods. For example, information, as found in a newspaper, is considered to be an experience good [SV99] – its value can only be assessed after reading it. However, if a consumer knows a certain newspaper to have high-quality articles, he or she will simply search for that particular newspaper using its brand name. Some drawbacks of this classification are that it assumes the consumer to be able to gain and assess the relevant information, measure the relevant criteria, as well as being conscious about the experience. For example, the effect of taking vitamin products may not be directly experienced. Other authors propose additional dimensions or categories, like credence [DK73]. For example, legal advice as an experience good may be additionally characterized by the level of trust in the skills of the advisor [Mef00].

Finally, a classification based on the physical/non-physical or material/ immaterial nature of a product, already mentioned in the context of services, distinguishes between *tangible* and *intangible* products [Lev81]. In general, tangible products have a physical appearance and can be touched, seen, smelled, or heard, whereas intangible products are not physical in nature. In practice, however, tangible products can also have intangible properties, as demonstrated by the example of the expensive pen (see Sect. 2.1.1). Software is considered as a typical example for an intangible or immaterial product [Bro95, KRS04]. Although software may produce outputs with physical properties on a screen or a printer, one has to bear in mind that such outputs are only different representations generated by the underlying software product.

2.2 Digital Products and Digital Information Products

The general remarks on products presented so far will provide the ground in this Section for the characterization of special types of products: *digital products*, and *digital information products* as a subcategory. As there is no general agreed-upon definition of what digital products are, various relevant definitions and opinions are presented next, and the main aspects are distilled in a working definition. Afterwards, detailed examples are provided, which are followed by an analysis of typical characteristics of digital products and classification of digital products. Finally, a model for value chain of digital products is introduced.

2.2.1 Definitions

Classic economic theory regards information only as a means to reduce uncertainty, rather than a tradable good. Therefore, it is proposed in [Loe00] to extend the view on information and digital content towards a digital product as "a production asset, a good, and a market attribute".

This attempt is supported by Wang et al. who advocate managing information as a product, in particular by using an analysis of information needs, well-defined production processes, or dedicated people like product managers [WLLS98]. Moreover, information products are assumed to have a life cycle with the stages introduction (creation), growth, maturity, and decline, which all need managerial attention in order to prevent a deterioration of information quality.

In a different approach, Varian views information goods as "anything that can be digitized – a book, a movie, a record, a telephone conversation" [Var00]. He emphasizes that the definition does not require the information to be actually digitized to constitute an information good. From this point of view, one can deduce a subset of digital products as digitized information goods.

Some authors like Leonhardt restrict the notion of digital products to the product data that represents the real (i.e., physical) product [Leo01]. However, this view is very restrictive and excludes for example software products which do not possess associated material products.

Choi et al.'s frequently cited definitions puts forward: "Anything that one can send and receive over the Internet has the potential to be a digital product" [CSW97]. From the context throughout their book, it can be deduced that the authors do not restrict themselves only to the Internet, but to computer networks in general. As an example, the authors consider the representation of information as a digital product. In particular, it should be possible to compose digital products out of text, data, graphics, video, or audio components. Furthermore, Choi et al. state: "All aspects of digital communication and processing can be considered to be digital products". Unfortunately, this definition is not specific enough.

Scupola focuses, in addition to electronic networks, on the value chain: "A digital product is defined as a product whose complete value chain can be implemented with the use of electronic networks, for example, it can be produced and distributed electronically [...]" [Scu03].

A different approach is taken by Hilbert by defining the complement: "Non-digital products must be physically delivered to consumers" [Hil03]. Furthermore, digital products should should be able to "bypass the [physical] transport and often even the wholesale and retail network".

Software in general is often regarded as an intangible, digital product [Gun78, KRS04, SHT04], and sometimes only if the associated documentation and data for delivery to the user is also included [IEE90].

In analogy to the general notion of services, Web services could be seen as a possible counterpart in the digital world (see [CD02] for details). However, Web services are defined only to provide standardized interfaces for enabling application-to-application communication [W3C07c]. Therefore, it seems inappropriate to consider a Web service as a digital product in its own right, since the actual product properties are essentially derived from the applications the Web service is connected to. It makes sense, however, to view the combination of a Web service and the connected applications as a product, in particular with the Web service as a product feature which offers an abstraction layer that hides the implementation, eases interoperability, and which provides composition possibilities to assemble digital products from other existing digital products.

The main aspects of the presented definitions for digital products are summarized as follows:

Definition 2.1 (Digital Product).
A digital product is a bundle of properties or features which are constituted by artifacts that are digitized or produced electronically. In addition, the bundle may have other properties which are intangible and not directly constituted by artifacts. Digital products can be distributed without loss in purely digital form (e.g., using computer networks, CDs, tapes, etc.). A digital product serves a specific purpose (i.e., it has a core benefit), is intended to be a tradable or exchangeable good, and can satisfy a want or need.

The definition covers many aspects of digital products in a general way. It characterizes digital products as bundles of properties. However, such a bundle should be divisible, since digital products typically consist of modular parts which should be easily replaceable, for example during version upgrades. Artifacts are understood in a way that they may have static properties (e.g., they store data) and/or dynamic properties (e.g., functionality in form of source code or executable programs). The artifacts constituting the product properties are digitized, i.e., information goods can be mapped to a digital representation, or the good is produced electronically right away. Consequently, the definition includes, for example, digital content, software and the corresponding source code files, files containing process specifications, the artifacts constituting a Web site, or files needed to implement a Web service. Furthermore, the definition allows other intangible properties to be part of the product bundle, which are not constituted by artifacts, like a well-known brand name, for example. The definition does make any restrictions on the architecture or organization of a digital product. For example, a digital product can be a monolithic, stand-alone entity, or have a distributed, client-server-based architecture where an artifact (client) draws upon data provided by another artifact (server). The digital product consisting of client/server-artifacts is merely using a different way to manage and organize its storage locations and functionality, compared to the product which is implemented as a stand-alone entity. Other important aspects of digital products are the possibility of lossless transportation, and that they remain in a digital form during the distribution process. Furthermore, a digital product has to serve a specific purpose and satisfy a want or need. Lastly, the bundle of properties becomes a digital product if and only if there is the intention to make it a tradable or exchangeable good, irrespective of the fact whether the good is offered for free or not, or whether it is financially successful. In the following it is assumed that digital products also have three conceptual levels of core benefit, actual product, and augmented product (cf. [KA05]), and that each of the properties in the bundle contributes some value to at least one of these levels.

Digital Information Products

Digital information products will be understood as a special category of digital products:

Definition 2.2 (Digital Information Product).
A digital information product is a special type of digital product whose core benefit is the delivery of information or education.

Building upon the widely-used definition by [LL03], *data* will be understood as "raw facts" which still have to be organized in order to be understood and used by people, and *information* as data that has been "shaped into a form that is meaningful and useful for human beings". The definition of information contains some subjective components (e.g., "meaningful" or "useful"), but this is inevitable since information is something that arises through the interpretation by human beings. In some cases, this subjectivity may propagate to the definition of digital information products.

An interesting aspect is that digital information products nowadays consist of a mixture of information (i.e., content) and software, and that the exact distinction between contained content and software begins to blur. There exists a spectrum with different degrees of mixtures where either the information aspect or software aspect prevail (see Fig. 2.3). On the one extreme, traditional software, such as a word processor, may for example incorporate a thesaurus or interactive help which have the properties of digital information products. On the other hand, Web pages are for example more content-centric and may contain also Java applets as additions; another example for a content-centric mixture is an executable program that contains and displays (possibly with a password protection) information in plain text. This thesis will concentrate on digital information products where the information aspect prevails (right side) and where the core benefit is the delivery of information. Further details on content are given in Sect. 2.2.5.

2.2.2 Examples

Digital information products are ubiquitous, which is underpinned by the following examples (see also [CSW97]):

- Electronic newspapers, magazines, journals, books, weather reports

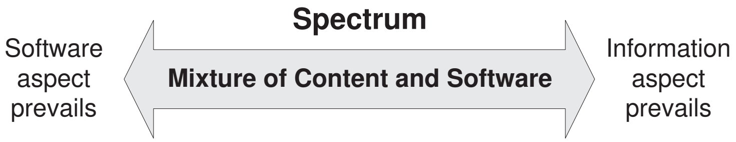
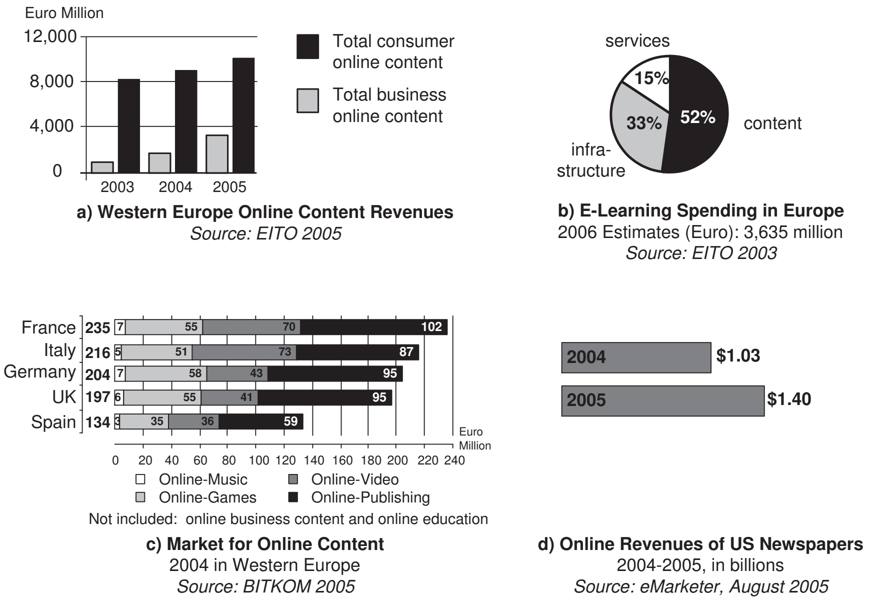


Fig. 2.3. A spectrum of digital information products.

- Digital (interactive) stock charts
- Electronic (interactive) travel maps
- Electronic phone books, yellow pages
- Digital audio: speech recordings, audio city guides (e.g., [Aud07]), podcasts (which represent published radio programs, spoken articles, or spoken news downloadable from the Internet, usually in MP3 format; see [KD05, Coc05])
- Digital video: movies (e.g., in AVI or MPEG format [KD05]), television programs
- Electronic user and training manuals
- Digitized lecture videos with annotations
- Learning content with interactive animations



a) Western Europe Online Content Revenues
Source: EITO 2005

b) E-Learning Spending in Europe
2006 Estimates (Euro): 3,635 million
Source: EITO 2003

Not included: online business content and online education

c) Market for Online Content
2004 in Western Europe
Source: BITKOM 2005

d) Online Revenues of US Newspapers
2004-2005, in billions
Source: eMarketer, August 2005

Fig. 2.4. Statistics related to digital products and digital information products. Sources: [EIT03, EIT05, eMa05, BIT05].

Figure 2.4 visualizes the economic importance of digital products and digital information products. There are estimations even claiming that over 90% of the information currently created is in digital format, and it is expected that much of the non-digital content will be digitized [Var05]. For statistics covering other aspects, the reader is referred to [LVS$^+$03, EIT03, EIT05, eMa05, BIT05, BIT06].

A digital newspaper

Digital products or digital information products which are the digitized counterparts of existing physical products, have the potential to offer new kinds of features which were previously impossible to deliver. For example, the core benefit of a digital version of a newspaper remains the provision of information (which qualifies it as a digital information product). In the actual product layer, however, the digital newspaper can provide more up-to-date information, or seamless integration of content from a variety of online sources. In addition, the digital newspaper can be equipped with interactive, full-text search capabilities, personalized information filtering, or clickable cross-reference links. The augmented product layer may offer additional index or table of contents alerts delivered regularly via e-mail.

Google Earth

The Google Earth system [Goo07] is a digital product offering access to a geographic database of the whole world, along with interactive 3D visualization capabilities. The core benefit of the product is primarily targeted towards the provision of orientation and navigation information, and covers educational aspects as well – it is, therefore, a digital information product. At the actual product level, satellite and aerial imagery are offered, which can be navigated using the mouse or exact coordinates. Other features are the possibility of overlaying other pictures (e.g., showing actual damages after a hurricane, historic sites, city maps). Depending on the price a consumer is prepared to pay, Google offers – in a product line – different features and different content, such as overlayed traffic information, risk information for insurances, real estate site analysis, environmental information, shipment and container tracking, or security information.

2.2.3 Characteristics of Digital Products

Digital products (and consequently, digital information products) have some characteristic properties which distinguish them from other types of products [CSW97]. The characteristics considered most relevant are discussed next.

Indestructibility

Digital products are indestructible in the sense that after they are created there is no wear and tear, so that a product could be used for ever [CSW97]. This has important implications, since a supplier of digital products will compete with his own products already sold in the past, and may quickly face saturation effects in the market. Furthermore, since digital products cannot be "consumed" just like physical products (i.e., the usage has no impact on the quality), a used digital product is equivalent to a new one. Therefore, to maximize profits, there is hardly any other escape except licensing the usage of a digital product (like charging a usage fee during a period while the product continues to exist), or providing different versions of it [Var97, SV99].

More specifically, versioning – which may provide different qualities of a digital product– can be done along a time dimension (e.g., the next release has some bugs removed) or a space dimension, where basically versions of the same product are offered at the same time with different sets of features. A more thorough economic analysis of the latter approach – which essentially results in a product line for digital products – has been done in the literature by Meyer et al. [MZ96] and Varian [Var97]. Several versions of the same product with different sets of features additionally have a diversification effect: the inexpensive version may generate revenue during recessions, while the most expensive one might bring additional profits during boom periods. Shapiro and Varian also attribute psychological effects to versioning, since for example from three possible product versions (e.g., light/basic/premium), undecided customers might preferably choose the basic version in the middle [SV99]. The versioning of digital products in a product line may be done, for example, w.r.t. time delay or quality of updated information, convenience of user interface, image resolution, supported file formats, comprehensiveness, programmatic features, annoyance, or support.

It has to be critically remarked that indestructibility, as understood in the literature [CSW97], assumes a well-working, secure environment. It neither considers hardware failures which may destroy the digital representation, nor harmful modifications by malicious software such as computer viruses.

Transmutability

Due to their digital nature, digital products are easy to modify. On the one hand, this is advantageous because digital products can be customized easily. On the other hand, this makes it extremely difficult to prevent unauthorized copying or modification to circumvent copyright protections.

Reproducibility

Digital products can be easily reproduced, transferred, or stored. This can be done without loss, so that there is no distinction between the original product and the duplicated one.

Cost Structure

As investigated in the literature [CSW97, Var00, KT02], digital products have a characteristic cost structure:

- There are high fixed costs for production and low marginal (or variable) costs for reproduction. In particular, the fixed costs that are usually incurred before production are not recoverable in case of failure, i.e., they are sunk.
- Due to the easy reproducibility, the marginal costs of production are almost zero. However, there are some exceptions where this does not apply, for example, if components with copyright fees are used, which are calculated on a per-unit basis.
- As the marginal costs are almost zero, the pricing of digital products cannot be based on the cost structure. It is generally agreed upon that more creative pricing is needed for digital products, for example by creating different versions for different customers or groups of customers, and charging the maximum price they are willing to pay (price discrimination).
- Digital products may have extreme economies of scale, since (apart from necessary hardware) there are almost no limits to the production of additional copies. Once the fixed costs are recovered, every additional unit sold represents almost pure profit.

The investigated properties have a direct influence on the way digital information products are developed, as will be shown later.

2.2.4 Classification of Digital Products

As there is no uniform definition of digital products in the literature, classification schemes for digital products either do not exist or have only limited applicability under the specific view of an author. For example, Hilbert introduces the "degree of digitality" notion to classify goods [Hil03]. He distinguishes between goods that will never be digitized (like food), goods highly susceptible for digitization which are hardly distributed physically (like software), and goods that will eventually get digitized, but face – from his point of view – technical or habitual obstacles at the moment (like movies). Other authors apply the search goods/experience goods classification (discussed in Sect. 2.1.2) also in the digital world [SV99]. It is also conceivable to use file types for categorization. However, the latter is still subjectively dependent on what information a user stores inside. Choi et al. have compiled a list of various criteria for a taxonomy that is objective and general enough to be in line with the notion of digital products discussed so far [CSW97]. The proposed criteria to categorize digital products are presented below:

1. **Transfer mode**
   This criterion refers to how digital products are obtained and how they access data. In particular, two sub-categories are distinguished:

- **Delivered products**
  Such products are downloaded from some location and become available in their entirety after download (e.g., a book in PDF format).
- **Interactive products**
  As an integral part of these products, they require (even after a download) a continuous connection and real-time communication with a server or other clients (e.g., interactive games, real time video-on-demand education).

2. **Timeliness**
   Timeliness measures the freshness and up-to-datedness of the information contained in a digital product. A digital product can be created to be time-independent or time-dependent.
   - **Time-independent**
     In a general sense, time has only a minor influence on the value of a digital product or the way the product is used.
   - **Time-dependent**
     Information is time-related, and has a significant influence on the usage of the product, which can be valued differently by different users. For example, for certain users information can become nearly worthless when it is out-of-date (e.g., news or stock quotes), while other users (e.g., scientists) may value it differently.
3. **Intensity in use**
   This distinguishes between how often a digital product is used and which utility can be expected.
   - **Single-use products**
     These products are used exactly once and are no longer needed after the intended purpose is served (e.g., a news collection for a particular day).
   - **Multiple-use products**
     Products in this category are used more than once, and the total utility can increase after every usage. However, the utility growth rate can have different shapes for different products. For example, an increasing growth rate may be related to a software product that can be operated more efficiently after many uses; a decreasing growth rate is imaginable for a computer game which becomes boring; a constant growth rate can be expected from the usage of a movie database
4. **Operational usage**
   This aspect is used to distinguish whether a digital product is a static document or an executable program. This is important since it can help

control the way a consumer uses a digital product.

5. **Focus on externalities**
   This criterion should characterize effects that are not directly attributable to the digital product itself, but which come along with its usage. The value of a digital product can be influenced depending on how many people use it. It is distinguished between:
   - **Positive-externality products**
     The value of these digital products increases with the number of users they are distributed to (e.g., chat rooms, online games).
   - **Negative-externality products**
     By contrast, the value of these digital products decreases with the number of users (e.g., exclusive news for stock market investors).

Although the presented criteria for a taxonomy of digital products may not cover any conceivable aspect, they simplify a comparison of existing digital products (which can be found, for example, on the Web), as well as the derivation of various strategic decisions. For example, delivered digital products could be changed to become interactive, in order to increase customer loyalty. To bypass the saturation problem when selling digital products with an infinite life span, the timeliness of a time-independent product can be artificially changed to become a time-dependent product. This way, outdated products will no longer be useful and leave more room for reselling. Another strategy might sell the up-to-date products for a fee and give the outdated products away for free – allowing potential customers to assess and experience the product (like an experience good). It might also be desirable to turn single-use products into multiple-use products. For example, news can be collected in archives for historic purposes; books, articles, or scientific papers can be collected in electronic libraries, adding a reference character to the collection as a whole. The operational usage of a digital product may help enforce copyright restrictions. A static document may not be able to enforce any restrictions. However, it is possible to deliver the document embedded into an executable program which restricts, for example, the viewing and printing capabilities. Finally, positive externalities can help boost sales for digital products. For example, vendors of shareware or freeware products may build up a customer base by giving away a basic version for free and charging for a premium version with additional features. Positive externalities are also vital for vendors of operating systems or software platforms, as an increasing number of users puts pressure on application developers to deliver compatible software – a process which, in turn, attracts new users (who benefit from a wider choice of compatible software) and creates entry barriers for other vendors of operating systems or software platforms.

2.2.5 The Value Chain for Digital Information Products

The discussion of digital products so far has not given details on how the value pertaining to the assumed product model layers consisting of core product, actual product, and augmented product, is created. It is generally agreed upon that the value or supply chain for digital products is itself also digitized [BW95, KR00]. To some extent, pure software can be a digital product if it satisfies the requirements of Def. 2.1 in Sect. 2.2. Therefore, typical software process models like the well-known waterfall model [Roy70], the V-model [Koo92, Bun04], the spiral model [Boe88], evolutionary development [Som04b], or the Rational Unified Process [IBM07] can be integrated into a value chain for digital products. In the following, however, this would lead to a too broad discussion and therefore, the main focus will be adjusted only to digital information products. Thus, further details on software development process models are not discussed here, and the reader is referred to [Som04b, Pre05] for an overview.

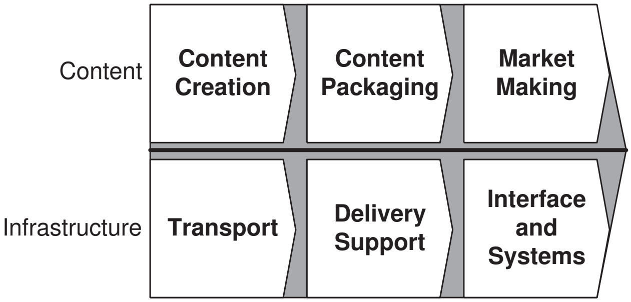


Fig. 2.5. Electronic publishing value chain [Eur97]

For digital information products, the value chain model put forward by the European Commission, originally developed for the electronic publishing (cf. Fig. 2.5), provides a reasonable abstraction which captures the most relevant activity areas for value creation [Eur97]. This model is also termed as the "2-3-6-concept" because it contains *two dimensions* (aligned horizontally) each of which is divided (vertically) into *three stages*, yielding a total of *six process areas* [SLS02]. The *content dimension* focuses on the creation of content, its packaging, and marketing. The *infrastructure dimension* covers communication and delivery aspects with the areas *transport*, *delivery and support*, and *interface and systems.*

Applied to digital information products, content creation includes the digitization of information goods or the creation of digital content. It must be emphasized that content nowadays consists of far more than of mere static text (in formats like ASCII, XML [HM04], HTML [W3C07a], MS-Word [Mic07]), but also of interactive animations (e.g., created with Macromedia

Flash [Mac03]), sound (e.g., WMV, MP3 [KD05]), video (e.g., AVI, MPEG [KD05]), or embedded programs (e.g., Java Applets [Sun07]) [OPS06]. In addition, the clear distinction between content and genuine software begins to blur, since embedding interactive functionality into content goes hand in hand with software development (see for example [PW98, DEH$^+$00]). Therefore, content creation is intertwined with software development, and thus details will be gradually developed and discussed throughout this work. After the content is available, it has to be packaged for distribution. This is usually done by adding so-called metadata (which is data describing data) like keywords, cataloguing information, abstracts, etc. In addition, content may be compressed or embedded into executable programs which act as containers or wrappers (similar to packaging in Fig. 2.1) that allow users to install it to a specified location. Executable programs may also help restrict the way content is used, for example, by disallowing printing. In the market making area, strategies are developed (e.g., using the classification introduced in Sect. 2.2.4) for availability of a digital information product, customer access, licensing, or pricing. Digital information products are frequently offered in electronic markets [CSW97] where customers can access, purchase and download digital information products from several suppliers, and which act as a medium for communication and distribution [1].

In the infrastructure dimension, the transport activity area deals with the transfer of the digital information product over electronic networks, whereas the delivery and support area targets activities such as Internet access, server platform management, bandwidth guarantees, or payment-process systems [SS97]. The interface and systems area provides the necessary hardware components (like terminal computers, Internet connectivity) as well as the software components for user interfaces. It should be mentioned that various types of specialized systems have evolved which can support different aspects of the work with digital information products, like for example *Document Management Systems (DMS)*, *Content Management Systems (CMS)*, or *Workflow Management Systems (WFMS)*. DMS are used to facilitate the activities occurring during the lifecycle of documents (which are, in fact, digital information products) like creation, storage, retrieval, publishing, modification, change and configuration management, or archival [GSMK04]. By contrast, CMS focus on the creation of content in collaborative and distributed environments and allow the assignment of roles (like author, editor, publisher, administrator) and responsibilities to different persons, i.e., the system imposes restrictions on who is allowed to do certain manipulations. The creation and presentation of content are often separated in that an author may fo-

[1] As an aside, it should be mentioned that there are several points of view of what an electronic market is, e.g., “networks that let customers compare and order offerings from competing suppliers” [MYB89], or a medium for price negotiation and completion of ordering transactions [Mer02]. As such details on electronic markets are not relevant for the further argumentation, the reader is referred to [DDL01] for an overview and discussion of different definitions.

cus on the creation of a text file, after which it is automatically processed by the system to have a predefined presentation layout and colors, and published in a specified location (e.g., on the Web). Various CMS and subtypes of CMS are presented in [Bau05]. WFMS emphasize the aspect of structured and repeatable collaborative work and usually have a predefined workflow model (i.e., a description of a real-world process which can be executed by computers [LR99]) which specifies how the work is performed by the actors involved (i.e., people or other computers). A workflow model specifies which digital products are created by which actors, as well as how and when they are forwarded to other actors. During execution, the WFMS creates instances of the workflow model which contain the specific paths taken by a particular digital product. An overview of WFMS and their history is given in [WK96]. It finally has to be noticed that many available systems implement an overlapping functionality using concepts from DMS, CMS, as well as WFMS. A detailed comparison is offered in [GSMK04]. Details on other related types of systems not discussed here as well as a vendor comparison can be found in [CMS05].

Overall, the presented value chain model represents a rough framework from which one can derive more detailed processes for the creation and distribution of digital information products. Furthermore, it can also be used as a means for analysis and categorization of suppliers in the market of digital information products, or as a means to develop positioning strategies. For example, [GSNHS03] use the presented framework to develop strategies in the context of scientific libraries, virtual universities, and e-learning.

2.3 Summary and Discussion

This Chapter is one important pillar for the analysis to follow. Based on different opinions found in the literature, it elaborates a definition for products in general, and introduces product classifications considered relevant for this work. Building upon this, digital products are presented as a special type of products with special properties. As a further specialization of digital products, digital information products have a core benefit restricted towards the delivery of information or education. Throughout the Chapter, a conceptual model with three layers – core product, actual product, and augmented product – is assumed to be underlying all types of products. The model for the value chain in electronic publishing is used to identify important areas for the creation of digital information products. One important insight of this Chapter is that the viability and success of digital information products in a commercial environment is highly dependent on their typical characteristics – which are different from traditional (physical) products. At the bottom line, the analysis reveals for digital information products that:

- from a market-based view, **versioning and the generation of variants are indispensable**.

- from a production-based view,
  - **production is difficult; reproduction is easy**.
  - the **fixed costs of production represent the biggest part** of the total production costs and have a significant influence on profits.

These insights and the realization that digital products can be time-consuming to create and maintain [Sch03a] motivate the methodologies to be developed in the next Chapters. The main questions to be answered are:

- How can different versions of a digital information product be efficiently created, managed, and reused?
- How can the fixed costs of the production of digital information products be reduced, using a systematic approach?

The research on software product lines (to be presented in Ch. 4) will provide an important basis for a solution that will address both questions.

3

Digital Information Products in the E-Learning Domain

This Chapter presents e-learning as a field in which digital information products are already intensively used. The main purpose is to provide a more profound understanding of the concepts discussed so far by focusing on this particular domain, and identify potential general problems which can be derived from various practical examples.

After a clarification of the notion of e-learning, the activity areas of the value chain for digital information products introduced in Fig. 2.5 will be exemplified in the context of e-learning. Starting with the infrastructure dimension, existing types of systems will be described. In the content dimension, the existing approaches of content creation, metadata and content packaging, and market making will be discussed. Based on the given details, the existing approaches will be evaluated to identify potential problems and open issues. Finally, a discussion of the results reveals the problem areas which will be further elaborated on in the thesis.

3.1 Notion of E-Learning

E-learning is a special form of distance education. Historically, distance education started already in the 1800s in a for-profit school in England which delivered courses by mail to rural residents, who thus were able to receive education independently of time and place [NM05]. With the development of technology, other media were used for delivery, like for example radio or television in the 1950s and 1960s, which are still used today. Later, in the 1990s, the ubiquitous availability of computers drove the development of computer-based training (CBT) with courses distributed on CD-ROM, while the Internet facilitated the delivery of Web-based training (WBT). WBT offered a wide range of new possibilities for synchronous learning (with same-time interaction between actors, independently of their locations) using for example audio and video conferencing, electronic whiteboards, screen sharing, instant messaging, chat, as well as possibilities for asynchronous learning (where both time and location of actors are different), like e-mail or discussion boards. Today, synchronous and asynchronous distance learning is often used in a mix with classroom learning, which is termed blended learning. In the following, e-learning will be understood as defined by the American Society for Training and Development (ASTD), a large association dedicated to workplace learning [KL05]: "E-learning (electronic learning): Term covering a wide set of applications and processes, such as Web-based learning, computer-based learning, virtual classrooms, and digital collaboration. It includes the delivery of content via Internet, intranet/extranet (LAN/WAN), audio- and videotape, satellite broadcast, interactive TV, CD-ROM, and more". Various aspects of other definitions related to e-learning are discussed extensively in [Kle02].

3.2 Infrastructure and Systems Architecture

Most of the current e-learning systems have a conceptual architecture which can be described using the reference model depicted in Fig. 3.1. In the reference model, people are supposed to play different roles. Using an authoring system, learning content is created by teachers or teaching assistants. The authoring system may interact with a Learning Management System (LMS), managed by administrators, to transfer content. An LMS is typically used to ease collaboration, workflow or administrative tasks, like for example, student registration, scheduling of events, storage and delivery of learning material, execution of online tests, or progress tracking [ACP01].

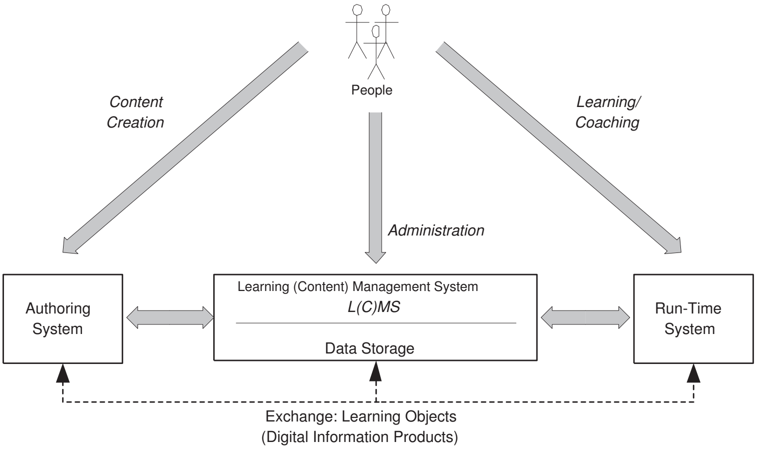


Fig. 3.1. A reference model for e-learning systems [OPS05].

Some LMS additionally offer content management functionality (cf. Sect. 2.2.5) along with standardized interfaces for content development and formatting, where the authoring component becomes itself part of the LMS. Such systems were termed Learning Content Management Systems (LCMS) [Bau05]. However, since current e-learning systems often provide features from both worlds, LMS and LCMS [HK03], this distinction is not further emphasized, and this is expressed by using the acronym L(C)MS instead [PSS04]. The data drawn from the L(C)MS is presented to the learners using a run-time system or module which implements a presentation layer. This module is actually used by learners for learning or by teachers for coaching. The technical realization of authoring system, L(C)MS, and run-time system may range for example from local, monolithical architectures, to Web-based, distributed architectures [HK03, TDN03] or to architectures focusing on mobile devices [TR03]. Examples of projects using Web-based L(C)MS are Web Life Long Learning (W3L) [BBZ04], Wirtschaftsinformatik Online (WINFOLine) [ESSW01], Virtual Global University (VGU) [PS05b]. Other e-learning reference models focusing on different aspects can be found in [BBSS01, DT06b, STBZ$^+$05].

Learning material usually has to be exchanged between the presented modules, as well as between different L(C)MS of different vendors. As a solution to this problem, the concept of Learning Objects was created, which is described next.

3.3 Learning Objects as Digital Information Products

The creation of learning material is expensive. For example, the ASTD offers as a rule of thumb 40 hours of development for 1 hour of classroom training, and 200 hours of development for 1 hour of computer-based training [NM05]; [Iss02] offers 625 hours of development for 1 hour of video-based training. Therefore, reuse of already existing material, as well as the interoperability and interchangeability of content between different L(C)MS, is an important issue which triggered the creation of so-called *Reusable Learning Objects* or *Learning Objects (LOs)*. According to the literature, it is possible that the term “Learning Object” was popularized by Wayne Hodgins in 1994 [Wil02, Pol03]. LOs are supposed to be packaged, reusable granules of learning content, annotated by standardized metadata to facilitate classification and retrieval. Beyond this agreement, there is no precise definition at the moment of what an LO is. For example, [Wil02] defines an LO as “any digital resource that can be reused to support learning”, while the IEEE LOM standard (which will be discussed in Sect. 3.3.2) defines an LO as “any entity – digital or non-digital – that may be used for learning, education, or training” [IEE02]. Furthermore, LOs have not much in common with object-oriented programming, since they mainly encapsulate data and only in a very limited sense methods [Kno04]. However, most of the definition attempts have in common that LOs satisfy all properties of digital information products (cf. Sect. 2.2).

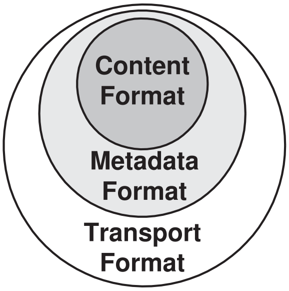


Fig. 3.2. An "onion model" of learning objects [PV05].

Although a precise definition of LOs is lacking, the technical structure of LO can be described using three conceptual levels, as depicted in Fig. 3.2. The onion model of learning objects distinguishes between the actual content, metadata about the content, and packaging information for transport purposes. With respect to what will be described in the following subsections, the core benefit of an LO is delivered through the learning content which is typically encoded in formats like (ASCII) text, MS-Powerpoint [Mic07], MS-Word [Mic07], Acrobat PDF [Ado05], HTML [W3C07a]. There are currently several metadata formats or guidelines for LOs (discussed in Sect. 3.3.2), like AICC, Dublin Core, ARIADNE, IEEE LOM, IMS, or SCORM CAM. As transport format, XML [HM04] or RDF [W3C07b] are widely used. The next subsections present more details for each of the layers in turn.

3.3.1 Content Creation

Learning content is typically created using authoring tools. Although in many cases LOs are regarded as traditional documents (e.g., a MS-Powerpoint file), various content models have been developed that specify an inherent, more general structure [VD04].

For example, the *Learnativity Content Model* [DH03] distinguishes between five different levels of content: *1) content assets* at the lowest granularity level represent raw media elements (e.g., sentences, paragraphs, pictures, audio and video files); *2) information objects* are sets of raw media elements and represent concepts, facts, or principles; information objects can be assembled to *3) learning objects* which serve a single learning objective, and which may include practice and assessment elements; *4) aggregate assemblies* are components with multiple objectives (e.g., chapters, lessons); *5) collections*, as a set of aggregate assemblies represent courses or curricula.

By contrast, the *Sharable Content Object Model (SCORM) Content Aggregation Model (CAM)* [DT06a] distinguishes between the levels: *1) assets* are a basic form of text, images, sound, etc; *2) Sharable Content Objects (SCO)* are a collection of one or more assets, and one of these assets must be "launchable" in the run-time environment of a SCORM-compliant LMS; *3) activities* represent meaningful units of instruction; *4) content organization* links activities to resources in 1) and 2); *5) content aggregation* describes a whole content package along with sequencing and navigation details.

In the *CISCO RLO/RIO* model [BL01], a *Reusable Learning Object (RLO)* has a fixed structure consisting of 7 ± 2 more fine-grained *Reusable Information Objects (RIO)*, and additionally, a part on overview, summary, and assessment. It is assumed that the complete RLO serves a single learning objective.

The *NETg* model [L'A97] has four levels of content types: *1) courses* may contain *2) units* that consist of *3) lessons* which themselves have several *4) topics*. A topic corresponds to an LO with a single learning objective, and it also has assessment parts.

Finally, it has to be remarked that the notion of an LO is used in different ways by the different models, which can be attributed to the non-existing general definition of LOs [Wil02]. The presented models are discussed and compared in more detail in [VD04].

3.3.2 Metadata and Packaging

At the moment, there exist a variety of metadata standards and recommendations in the area of e-learning, which also cause confusion and misunderstanding. As depicted in Fig. 3.3 a), some metadata standards also use parts of other standards, for example by selecting a concept, or a subset of metadata elements and related value sets (often called an "application profile" [NDTN03]). This somewhat intricate situation is not only the result of a lack of consensus, but is also influenced by the way committees collaborate to create e-learning standards [Duv04]. As shown in Fig. 3.3 b), there are basically two camps: organizations or consortia who produce specifications, but who do not have the authority to create standards (e.g., the main players IMS [IMS07], ARIADNE[ARI07] , ADL [DT06b], AICC [Avi07]), and accredited organizations who may use the specifications to create standards meeting the needs of the whole domain (e.g., CEN [CEN07], IEEE LTSC [IEE05] , ISO/JTC1/IEC [ISO05]). The most important links between standards and committees are explained next.

The intention of the *Aviation Industry CBT (Computer-Based Training) Committee (AICC)* is to "assist airplane operators in development of guidelines which promote the economic and effective implementation of computer-based training (CBT)" [Avi07]. The guidelines and recommendations, e.g., for interoperability, Computer Managed Instruction (CMI), or testing procedures, are general enough to be adopted beyond their initial scope. The AICC coordinates its efforts with IEEE LTSC, ADL, and IMS [HC00].

The *Dublin Core (DC)* initiative defines a small set of metadata elements, originally used for content in digital libraries, for the description of

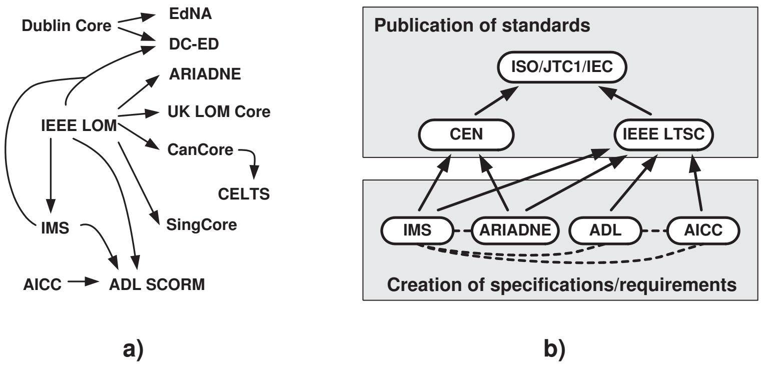


Fig. 3.3. a) Metadata standards/recommendations for learning objects; b) Collaboration for standardization in e-learning between the leading committees; based on [NHHMA03, Duv04].

LOs [ISO03a]. There are 15 elements related to content (*title, subject, description)*, intellectual property *(creator, publisher, contributor, rights)*, and document instances *(date, type, format, identifier, source, language, relation, coverage)* [PV05]. Additionally, an abstract model can be used which specifies how to create controlled vocabularies following encoding guidelines for the elements. Very often, DC elements are embedded into HTML (via meta tags), XML, or RDF. There are also parsers for automatic extraction of some elements from Web pages (e.g., [Des07]). Various extensions of DC have been proposed, which have additional descriptors for e-learning resources, like for example *DC-ED* which was proposed in a working group of DC, and which also draws upon IEEE LOM and IMS (described later). The *EdNA (Education Network Australia) Metadata Standard* [EdN06] is another standard which builds upon DC. Concerning the element set of DC, it has to be critically remarked that an empirical study with more than 910,000 analyzed records revealed that some elements are heavily used (*e.g., creator, identifier, title, date, type* were used in more than 70% of the cases), while there was almost no usage of the remaining elements [War03].

The *Alliance of Remote Instructional Authoring and Distribution Networks for Europe (ARIADNE)* [NDTN03] is involved in the creation of metadata standards as well as in the provision of a "Knowledge Pool System" as a distributed repository of LOs. The ARIADNE metadata has data elements divided into six categories: *general, semantics, pedagogical, technical, indexation, and annotations.* These were used as a starting point for the IEEE LOM standard (described later). At the moment, the ARIADNE metadata is a subset of IEEE LOM (see [NDTN03] for details on mappings). An empirical study of 3700 metadata instances collected during 7 years in the ARIADNE Knowledge Pool System shows that many elements are used only in about 50% of the cases, while other are hardly ever used. [NTD03]. In addition, LOs are most frequently (in decreasing order) of type narrative text, exercise, hypertext, slides, self assessment, questionnaires and videos, while the most common file types are HTML, PDF, MS-Word, and MS-Powerpoint.

The working groups of the *Learning Technology Standards Committee (LTSC)* of the *Institute of Electrical and Electronics Engineers (IEEE)* defined several standards covering various aspects of e-learning systems, which are for example related to architecture, digital rights management, computer managed instruction, or *Learning Object Metadata (LOM)* [IEE02]. The metadata specification of LOM has more than 60 elements divided into nine categories: *1) general (describing an LO as a whole), 2) lifecycle (related to the evolution of an LO), 3) meta-metadata (information about the metadata), 4) technical (technical characteristics of an LO), 5) educational (pedagogical characteristics of an LO), 6) rights (property rights and terms of use), 7) relation (specification of "links" between LOs), 8) annotation (change information), 9) classification (easing retrieval).* All elements defined in this standard are optional, and the standard even allows different levels of "conformity" (i.e., customized extensions of elements are possible) which may lead to interoperability problems in practical situations [PV05]. Whenever possible, the standard uses controlled vocabularies (i.e., a predefined value space) for the attribute domains, and parts of it can be mapped onto Dublin Core [IEE02].

A survey conducted by the ISO/IEC committee has analyzed 250 instances of different repositories with respect to the usage of LOM elements [Fri04a]. It found out that the vocabularies used were inconsistent, and that only few of the elements were used frequently, while the remaining ones were hardly ever used. In particular, most of the elements in the *educational* category were used in less than 30% of the cases, which is paradoxical for a standard intended for the description of educational resources [POS05]. These findings raise the suspicion that there are too many elements in LOM, and may help explain why there are so many adaptations or application profiles of LOM [DH03]. Various other metadata standards are based on LOM: ARIADNE, UK LOM Core [UK 07], CanCore [Can04], or SingCore [Ng01]. CELTS, in turn, is a subset of CanCore [ISO03b].

The *IMS Global Learning Consortium*, originally initiated as the *Instructional Management System (IMS)* project, develops specifications for different areas of e-learning, like for example content packaging and sequencing, or question and test interoperability (there are more than 20 areas, see [IMS07]). Compared with other specifications, the *IMS Content Packaging* defines more details of the structure of an LO, with respect to "content resource aggregation, course organization, and meta-data" [IMS04]. In principle, an LO is seen as a stand-alone package of content that "can be delivered independently, as an entire course or as a collection of courses"(cf. Fig. 3.4). The package consists of *physical files* (e.g., files of type MS-Powerpoint, PDF, MS-Word, HTML) and a *manifest.* The manifest contains *metadata* (e.g., in IEEE LOM format), an *organizations section* describing possible course outlines and an order of logical resources, a *resources section* defining available logical resources and their connection to physical files, and (optionally) other *sub-manifests*. Other details on IMS are discussed in [PA01, Duv04].

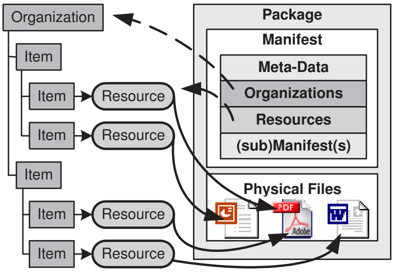


Fig. 3.4. IMS Content Packaging

Finally, the *Advanced Distributed Learning (ADL) Sharable Content Object Model (SCORM)*[DT06b] follows a different strategy: it tries to integrate the best of other standards into one standard. The proposed *Content Aggregation Model (CAM)* [DT06a] uses metadata specifications from IEEE LOM, content structure derived from AICC, and sequencing and content packaging from IMS. SCORM-compliant L(C)MS have to implement an *activity tree*, which is a data structure describing the conceptual content structure. For each learner, learning resources are mapped to the nodes of the tree, which represent learning activities. In addition, SCORM describes the conceptual structure of run-time environments for LOs with respect to interfaces, communication mechanisms, or temporal session models. SCORM is widely discussed in the literature, for example in [HC00, PA01, NHHMA03, Duv04, VD04].

3.3.3 Market Making

E-learning is on the verge of becoming a business, and education is perceived to be probably the world's largest information industry [HWV96]. Electronic markets for sharing of LOs are beginning to emerge in form of Web-based repositories, like for example ARIADNE [ARI07], CAREO [CAR07], Merlot [MER07], EdNA Online [EdN07], EducaNext [LMQS03], Smete [SME07], or Intute [Int07]. With respect to the classification introduced in Sect. 2.2.4, the LOs found in the repositories are mostly delivered (although a few might have interactive components with synchronous communication capabilities); many LOs are time-independent, as they consist in many cases of simple files (see also studies mentioned in Sect. 3.3.2); LOs are intended for multiple use; LOs can be or contain static documents or executable programs (e.g., based on Java or Macromedia Flash [Mac03]; see also [BMB04]); some LOs using components supporting synchronous communication may have positive externalities as the user base grows.

However, there are still many obstacles to overcome, like for example quality problems [Bos03, Som04a], or a closed marketplace with limited opportunities for selection [Dow03]. An empirical study of the e-learning suppliers' market in Europe found out that the market for learning content is still very immature and that it actually consists of several segments (higher education, workplace learning, vocational education and training, schools segment, home segment) [Dan05]. Furthermore, most e-learning is delivered as blended learning to on-campus students. It also seems to be difficult to motivate teachers to share LOs freely and ensure that LOs are fully interoperable. In the overall market there is little revenue growth, which makes competition tough. The market participants thus have to drastically reduce costs in order to be profitable. In addition, viable business models still have to be found, which is also an area of research [EER02, SW05].

3.3.4 Evaluation of the Concepts behind Learning Objects

Despite the fast-paced developments in the e-learning field, many conceptual aspects related to LOs and reuse of courseware are still in their infancy [Dod02], which contradicts the wide-spread belief that the current problems in e-learning are only a matter of better standardization. This becomes obvious during the practical creation of courses based on LOs, which reveals that suitable LOs are hard to find in repositories, and that LOs obtained from repositories often do not "fit" together because of a mismatch with respect to file type, structure, layout, didactics, or presentation (see also empirical case-studies in [CA04]). Furthermore, metadata standards in e-learning do not try to integrate the metadata already contained in various file types, like for example in PDF, MS-Powerpoint, MS-Word, or MPEG. An evaluation of technical, didactic, and economic problems of LOs are discussed next, and the possible roots of the problems are outlined.

Technical Aspects

The LO paradigm has much in common with *component-based software development*, which is a reuse technique whose idea is to build software from existing tradable and exchangeable components rather than from scratch; similarly, in the LO paradigm, e-learning courses are constructed using LOs as components. This technical similarity is used here to evaluate the LO paradigm from the perspective of component-based software development, as the latter has similar problems to solve and obstacles to overcome. Moreover, it has been already mentioned that the boundaries between content and software begin to blur. Compared to the relatively young developments in the LO area, the field of component-based software development is more than 35 years old (see [McI68] for an initial paper) and offers theoretical concepts and empirical findings which are highly valuable for LOs. A software component is generally understood as “a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties” [SGM02]. In addition, software components are modular artifacts whose state is not observable from outside, and therefore context dependencies between a component and other components or systems have to be expressed explicitly.

Based on the experiences made with component-based software development, Fig. 3.5 shows the main categories of requirements for a successful use of components [Aßm03]: 1) a *component model* is needed to describe precisely how components might look like and when two components are exchangeable; 2) *composition techniques* provide the means to combine or adapt components; 3) a *composition language* is needed to specify how to build systems out of components, for example by describing the corresponding architecture.

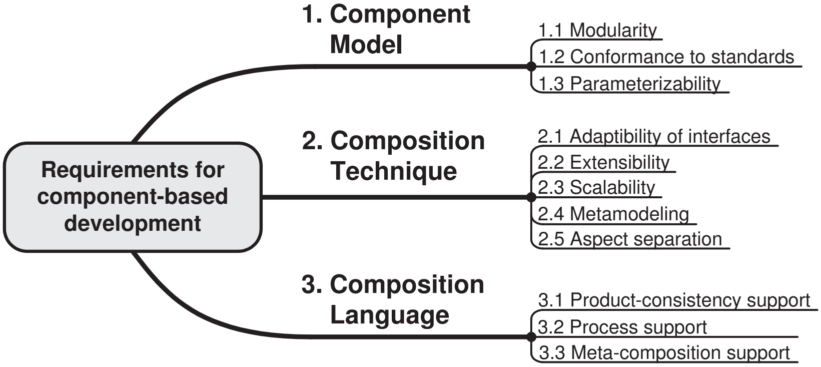


Fig. 3.5. Requirements for component-based development; based on [Aßm03, POS05].

Due to the abundance of specifications in e-learning, there is currently no uniform component model for LOs (see also [Fri04b, Pol03]). In addition, some specifications (e.g., IEEE LOM) do not provide precise models at all. For software components, the *modularity* of components is realized by following the information hiding principle along with the definition of interfaces. Very often, additional specifications or pre- or post-condition constraints (i.e., contracts between modules) are necessary to define how software components should work together, e.g., an operation defined in one interface may need an operation defined in another interface (see for [SGM02] details). For LOs in e-learning, modularity is satisfied, although more on a logical level, since LOs mostly consist of a loose collection of files and the package structure is often defined using additional metadata. However, LOs are not conceived to mutually use data or functionality from other LOs. Excluding a few exceptions

[SS03], contract specifications for LOs are not in the focus of attention. The *conformance to standards* should make components exchangeable on component markets. In general, it is important to have a small number of standards to ensure an acceptable market share for one standard [SGM02]. It has been even shown that there are situations in which overall costs with standardization may increase for all participants if a standard fails to have a critical mass of users [BK98], and that network externalities play an important role [FS86]. The great number of standards for LOs in fact complicates their exchange and reuse. In addition, the *parameterizability* and adaptability of existing LOs is very limited. Some frequently used file formats (like PDF, Flash, Java bytecode without source files [Sun07]) often allow LOs to be reused only in the way they are delivered, and do not allow easy modifications and adaptations to the reuse context.

There are similar shortcomings related to the composition techniques for LOs. Without precise models or interface specifications, it is difficult to define general techniques for LO composition. As with parameterizability, the *extensibility* or *scalability* of LOs can be very limited, depending on the file types used. *Metamodelling*, which eases the analysis of composition techniques for LOs, is currently considered only on a rudimentary level (e.g., meta-metadata in IEEE LOM). *Aspect separation* for LOs is an area of research [Pan05]. In software development, aspects are crosscutting concerns, i.e., parts of code which logically belong together, but which are physically distributed across different modules because of the chosen decomposition criteria [1] (e.g., code for failure handling or synchronization) [FECA04]. The concept of aspects can be used similarly in e-learning; for example, a layout specification can be treated as an aspect that occurs in several LOs. Then, during the packaging process of LOs, the aspects are woven into their predefined locations, shortly before LOs are delivered (see [Pan05, PV05] for details).

Furthermore, there are no wide-spread, general composition languages for LOs, which is probably due to the inadequate component models and composition techniques. Although the concept of dynamic course generation has been researched in some systems [Ate04, FLT04], course assembly out of LOs must often be done manually, because only this ensures the *product-consistency* with respect to file type, structure, layout, didactics, or presentation. Furthermore, a composition language for LOs should be easy to integrate into the overall development process, and should support meta-composition (i.e., it should itself be based on a component paradigm). Other details on the comparison between LOs and software components omitted here are presented in [POS05].

Some research already attacks a part of the presented technical problems at different levels. For example, Web services are used as a means to package LOs which, in addition to data, provide interfaces and functionality that

[1] It is assumed that there are situations in which there is no other way to decompose a system into modules which would cleanly encapsulate all concerns.

may process and deliver the data contained in LOs [PV03, PSS04]. Other reuse techniques, like design patterns for LOs, try to ease the adaptability in different reuse contexts by encoding the requirements for reuse in a pattern definition [Jon04]. The empirical studies of the usage of the various standards mentioned in Sect. 3.3.2 suggest that a general adequate abstraction for the description of LOs has not been found so far.

Didactic Aspects

Most of the e-learning standards are designed to be “pedagogically neutral”, i.e., the design and usage of LOs is independent of the chosen pedagogical approach. For example, SCORM is criticized to be “limited in its pedagogical scope, neutrality or relevance”, and to support only single-learner scenarios [Fri04b]. Because of such reasons, other approaches like the Educational Modeling Language (EML), Learning Roles, Essener-Lern-Modell, IMS Learning Design, DIN didactic object model, or didactic ontologies try to fill this gap and provide formalized didactic descriptions for instruction (see [TS04] for a detailed overview). Furthermore, due to the support of committees like ADL by the US Department of Defense, LOs and e-learning standardization are suspected to “bear the imprint of the ideology and culture of the American military-industrial complex” which imposes a “military worldview”, although the “goals of public education” are claimed to be “radically different than those of the American military.” [Fri04b]. Finally, the context of content reuse should not be fully neglected during LO design [Eur04]. Related problems, like for example on social problems, are collected in [Mor00].

Economic Aspects

It remains generally unclear whether interoperability and uncomplicated migration of LOs between different L(C)MS is really desired by commercial vendors. Assuming that there was a single, wide-spread standard with an adequate model for LOs, the only competitive advantage would be constituted by the learning content itself, since everything else would become easily exchangeable. However, content is expensive to produce and the investment has to be protected. In such situations, vendors would try to build up entry barriers for competitors (e.g., by creating high quality content which is bound to a proprietary platform under their control), and exit barriers for existing customers (e.g., by making switching to other platforms difficult through complicated migration and reduced interoperability) [PSV05, Mor00, Eur04]. In fact, [Sie04] claims that the “real issue is that LMS vendors are attempting to position their tools as the center-point for elearning”, not the content. In addition to the technical shortcomings, these aspects may also explain why there are “hundreds of academic or commercial platforms [...] present on the e-Learning market”, and why “[m]ost of these systems are closed and allow neither share nor re-use” [VBDT04], or why the “reality of elearning is a hodge-podge of legacy repositories, protocols, special interest groups and self-serving communities" [HREW04]. It is also ironically recognized that standardization in e-learning is meeting the needs of "vendors with the content they have right now" [Fri04b].

3.4 Summary and Discussion

This Chapter presents e-learning as an example of a field in which digital information products play an important role. The content and infrastructure dimensions of the 2-3-6 concept describing the value chain of digital information products (see Sect. 2.2.5) are presented with more details for this special field. In the infrastructure dimension, it is shown that L(C)MS have evolved as a special kind of systems for e-learning. In the content dimension, it is explained that LOs are in fact digital information products using possibly different data formats for content, metadata/packaging, and transport. However, the described abundance of standards and recommendations should also convey the current confusion in the e-learning field. More and more participants are beginning to realize that the complexity and difficulty of many problems were underestimated, and that "the use of learning objects was not nearly so simple as [...] at first assumed" [Dow04]. This is supported by the analysis of the concepts behind LOs, which reveals that component models of LOs are not well-developed, didactic aspects partly neglected, and that there are economic reasons that may complicate interoperability and migration. In conclusion, it can be derived that:

- from a market-based view, the **learning content constitutes an important competitive advantage** for digital information products in e-learning.
- from a production-based view,
  - **better reuse techniques are required for learning objects** as digital information products. Such techniques should help cut the fixed costs of production, which is vitally important in the current low-margin markets.
  - beyond reuse on a "code-and-file-level", **reuse techniques for LOs also need to target higher conceptual levels** (e.g., ensuring identical component models for related LOs, modeling of similarities between sets of LOs, or modeling the reuse context of several related LOs)

This is underlined in a report of the European Commission which states that indeed "[n]ew models are required for selecting, producing, using and re-using content for elearning" [Eur04]. The wide-spread belief that interoperability and reuse problems will be solved as soon as the current standards mature and converge seems elusive [PS06]. Based on the assumption that content of digital information products can be just as complex as genuine software, the reusability problems with LOs are likely to persist beyond the present standardization discussion. In addition, the time of adoption can play a role as well [FS86, BK98].

The illustrated parallels to Software Engineering show that reuse is a complex issue which cannot be solved satisfactorily in a general way [SGM02, POS05], and that it “is far too simplistic to assume that components are simply selected from catalogs, thrown together, and magic happens” [SGM02] – which seems to be true also for digital information products. A situation with several standards is also typical for domains other than e-learning, such as for example digital libraries [ESK04, TL05, KD05]. Very often, “successful and efficient reuse involves constraints and tradeoffs, and requires planning and coordination in advance before reusable artifacts are created” [OPS06]. The next Chapter presents an overview of software product lines as a reuse technique that follows this approach, and which will be extended later to manage the reuse context and commonalities of digital information products more efficiently.

4

Software Product Lines

This Chapter presents software product lines as an approach to software reuse, which draws upon principles used in today's manufacturing product lines. As an overview, it presents the currently most influential methods and techniques as well as some open research questions in a generic way which is independent of individual approaches, along with the key contributions of different approaches.

After an introduction that sketches traditional approaches to software reuse with their intentions and drawbacks, the approach of software product lines and its underlying hypotheses are described. Thereafter, three relevant sub-processes of the overall product line development process are introduced in a generic way: domain engineering with the view on all products in a product line, application engineering with the view on a single product, which is derived from the results of domain engineering, and family engineering with the view on the family itself with respect to organizational issues. For domain engineering, details are outlined that are relevant for domain analysis, domain design, domain implementation, and domain testing. The same is done for application engineering with application analysis, application design, application implementation, and application testing. Family engineering is understood in the sequel to subsume, for a product family as a whole, economic aspects, organizational aspects, and evolution and maintenance aspects. Being aware that the presented methods differ in their details, the main concepts of a software product line are finally distilled into a more precise definition that will serve as a reference in the following Chapters. A final discussion summarizes the main insights.

4.1 Traditional Approaches to Software Reuse

This Section outlines some of the well-known state-of-the-art approaches to software reuse on different abstraction levels, along with their intentions and drawbacks.

Object-Oriented Programming

Object-oriented programming languages introduce language constructs like inheritance to support reuse on the code level. Additional flexibility is provided by different types of polymorphism [CW85]. The reuse of code by programmers, however, often occurs on a fine-grained scale and in a rather opportunistic way without coordination in advance. It has also been noted that excessive use of inheritance can lead to designs that are more difficult to reuse [GHJV95]. A copy-and-paste approach for reuse of code by programmers, sometimes referred to as "code scavenging" [Kru92], has proven not to be effective for successful reuse [Bos00].

Design Patterns

In the context of object-oriented software design, patterns are understood to be a general, proven solution for a recurring design problem [GHJV95]. They are intended to be an extensible template describing a generalized solution on a higher abstraction level (i.e., design level instead of code level). In object-oriented languages, design patterns can show relationships and interactions between classes and objects. An important progress is that patterns are able to capture design knowledge and model a context of reuse. One drawback is the lacking capability to describe design on higher levels of abstraction.

Libraries and Frameworks

During the construction of larger software systems, frequently needed functions or subprograms can be stored in a separate collection which forms a library. As a further development, object-oriented frameworks attempt to provide an abstract design for a particular domain [Bos00]. A framework is understood as "a set of classes that embodies an abstract design for solutions to a family of related problems" [JF88]. In contrast to design patterns, a framework is more specialized and addresses all or only a part of an application. For example, a framework for a graphical user interface can be an almost finished part of an application, which can be extended and customized. Based on the extension mechanisms, it is distinguished between white-box (inheritance-based) and black-box (parameterized) frameworks [JF88]. Frameworks can also be classified into calling (i.e., they completely control the application and actively call other parts) and called frameworks (i.e., they are invoked by the application) [SBF96]. Frameworks facilitate a reuse of architecture and make predefined assumptions about their context of reuse. However, this can also lead to problems if more than one framework has to be used in an application [Bos00].

Components

The idea of software components has been discussed in Sect. 3.3.4 in the context of learning objects. From the reuse perspective, software components are still perceived to have several drawbacks. For example, component technologies are heavily dependent on specific platforms or middleware and have strong ties to the implementation technology [AB05, GSC$^+$04]. In addition, it is hard to predict the properties of software that is built from retrieved components, since the technologies for component specification and packaging have difficulties to capture in general the information on how components interact and depend on each other, how they function, or what resources they may consume [GSC$^+$04]. Furthermore, the idea of components as "LEGO bricks" seems elusive, since during the combination of components, further adaptations may be necessary [Bos00].

Architectural Styles

The abstraction level can be raised further to describe the architecture of a system from a more general perspective which encompasses components, as well as their interactions. An architecture of a software system defines that concrete system in terms of computational components and interactions among those components [SG96]. An architectural style defines "a family of [...] systems in terms of a pattern of structural organization. More specifically, an architectural style defines a vocabulary of components and connector types, and a set of constraints on how they can be combined" [SG96]. For example, in a pipe-and-filter style, components have input and output streams on which they operate, and streams are directed between components using pipes as connectors. As a more recent approach, Web services [CD02, W3C07c] have made service-oriented architectures popular. Architecture description languages have been developed as a means to formally capture architectures. However, their capabilities to describe or analyze general architectures can vary widely [Cle96].

4.2 The Approach of Software Product Lines

The lessons learnt from the experiences with software reuse so far are that without proper planning, the costs with reuse can be higher than developing from scratch [BKPS04], and even other criteria like software quality can suffer when reuse occurs in unintended or uncoordinated ways. Some of the root problems of reuse are for example excessive generality, a too broad scope, one-off development (without paying attention to reusability), or process immaturity [GSC$^+$04]. The need to reuse processes is also emphasized in [Rom05], and the importance of an overall strategy for reuse is also stressed in [Bos00].

Software product lines are a proactive approach that tackles the reuse problem by planning in advance, reducing the scope to a predefined domain, and exploiting commonalities between a set of related software products in the domain. The notion of a product line is borrowed from marketing, where it is understood to be a "group of products that are closely related because they function in a similar manner, are sold to the same customer groups, are marketed through the same types of outlets, or fall within given price ranges" [KA05]. Thus, the relation and similarity between products in a product line can be based on different criteria: similar technical aspects, customer groups, distribution channels, or similar pricing. Especially technical aspects are widely used to create product lines. For example, it is general practice in the car manufacturing industry to create common platforms (e.g., for chassis, electrical equipment, etc.; see [RU98]) for different brands of cars, and to limit the number of possible customizations in advance. The tradeoff is that customers will only be able to choose from a limited number of colors, seat types, or engines. However, such practices lead to significant cost savings and at the same time improve reuse of parts through a coordinated reduction of variability. Variability is often informally understood as the number of possible configurations or adaptations. Product lines try to find a balance between the degree of individual customization and the efficiency of creation of products.

Although it has been realized already in the 1970s that these principles can be applied to software [Dij72, Par01], this area has not received much attention until recently, because the reuse approaches presented in Sect. 4.1 were believed to be adequate solutions by themselves. Although in the beginning a distinction has been made between the notion of software product lines and software product families, this distinction is not emphasized any longer and the terms are frequently used interchangeably [vdL02, CN02]. From a general perspective, the definition of [CN02], which has been widely accepted by the community, will be adopted in the following:

Definition 4.1 (Software Product Line).
A software product line 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 and that are developed from a common set of core assets in a prescribed way.

An important point in the definition is the focus on a set of related systems instead of single systems, which requires a longer-term strategy. In addition, the scope of the examined domain is reduced to a particular market segment or mission, i.e., the view is not general, but specialized on a particular area of expertise. In contrast to the marketing literature (e.g., [KA05]) where the product line notion does not require products in the product line to have technical commonalities, the software product line notion explicitly demands for technical commonalities between the software systems in one product line. Furthermore, a system is built only from a fixed set of predefined core assets which can be (possibly parameterizable) artifacts. For example, there can be different types of core assets like architecture definitions, reusable software components, domain models, schedules, budget plans, test plans, or process descriptions [McG01, CN02]. A concrete software system is finally created by binding possible parameters of core assets to permitted values, and using them in an assembly process which is defined in advance in a production plan.

A feature is generally understood as a "requirement or characteristic that is provided by one or more members of the software product line" [Gom05], and as a "a logical unit of behavior" which embodies functional as well as non-functional requirements (e.g., quality) [Bos00]. The approach of software product lines is a proactive approach to control evolution and coordinate reuse, which defines in advance all possible product features and allowed feature combinations, as well as the assets used to implement the features. This leads to savings across the family of related products (economies of scope).

The reuse approach of software product lines is based on three underlying hypotheses[WL99]:

- **Redevelopment hypothesis**
  Most software development consists of creating variations of existing systems. Among the created systems, a major part is identical in any system, and the variations which make up the differences are – in comparison – small. Thus, the redevelopment of common parts is avoided.
- **Oracle hypothesis**
  The variability and the changes to a system that are likely needed are predictable (otherwise, a product line may not be profitable [BKPS04]).
- **Organizational hypothesis**
  The organization as well as the software itself can take advantage of the predictability.

Based on experience [PBvdL05] and these hypotheses, the general development process is split up into two sub-processes which are called domain engineering and application engineering[1] (cf. Fig. 4.1). In *domain engineering*, a domain engineer creates a definition of the whole product line. At the moment,

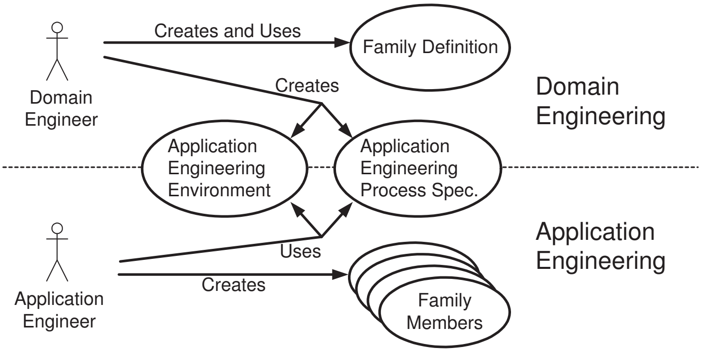


Fig. 4.1. Artifacts in Domain and Application Engineering (based on [WL99]).

[1] In this context, the word "engineering" is typically used in a narrower sense than traditionally (cf. [Web95]).

this is frequently done by creating a model which specifies all possible features for all products in the product line, along with restrictions that express valid feature configurations. In addition, the commonalities and variabilities of all future products are analyzed. The resulting choices are fixed and incorporated into the model, which helps to delay some of the design decisions until application engineering [AB05]. Using the family definition, a domain engineer creates an application engineering environment which contains the necessary tools as well as the artifacts and parameterizable core assets needed to implement the features of the family definition. The designed application engineering process specification (sometimes also called a production plan [CM02]) specifies how to exactly combine core assets to obtain a software product which may implement a subset of the predefined features. During *application engineering*, an application engineer is supposed to use the application engineering environment with its artifacts and tools, and to produce software products, which are actually members of the same family. Since reuse was planned in advance, there is no time-consuming and costly search for suitable components. Of course, an application engineer might perform custom additions and adaptations on a product, if demanded by a specific customer or market segment, as long as the resulting software product does not violate the constraints of domain engineering.

Further connections between the domain engineering process and the application engineering process are shown in Fig. 4.2. The philosophy of software product lines is that the major investment has to be made throughout the

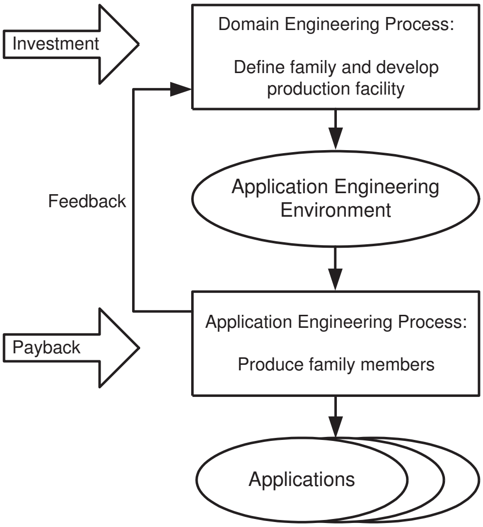

Investment
Domain Engineering Process:
Define family and develop production facility
Feedback
Application Engineering Environment
Application Engineering Process:
Produce family members
Payback
Applications

Fig. 4.2. Connections between the Domain Engineering and the Application Engineering process (based on [WL99]); ovals represent artifacts resulting from the processes that are represented as rectangles

domain engineering process which basically focuses the product line on a particular area, narrows the angle of vision to the relevant details, analyzes and determines valid choices, and prepares accordingly the reusable artifacts and tools which go into the application engineering environment. During the application engineering process, the choices are bound to concrete values which must be within the ranges defined in domain engineering. Since the concrete software applications are created in a generative manner [CE00], there are fewer activities and shorter development times – the payback of the investment in domain engineering. The feedback loop from application to domain engineering enables a continuous evolution and enhancement of the domain models.

In addition to the two sub-processes of the development process presented so far – domain and application engineering – there is a third sub-process cutting across these processes, which is often neglected in the literature. This process is sometimes simply called “management” [CN02] and the activities performed within this process often depend on the specific view of the respective authors. In the following, *family engineering* will be understood to have the view on the family itself, and especially on organizational and economic aspects of software product lines.

The discussion so far introduced the main concepts on a macroscopic level. In the literature, various approaches refine the concepts in different ways [vdL02, BKPS04, PBvdL05]. The relevant state-of-the-art methods and models currently used in each of the sub-processes are presented next in a generic manner, which is independent of a specific approach.

4.3 Domain Engineering

Domain engineering is understood to be a sub-process of the overall development process which focuses on issues concerning all products in a product line. In particular, [PBvdL05] define it as follows:

Definition 4.2 (Domain Engineering).
Domain engineering is the process of software product line engineering in which the commonality and the variability of the product line are defined and realised.

A domain is understood to be an artificial construct which restricts the attention to one or more particular areas of interest [Ara94]. An important assumption is that the considered domain should be “stable”, i.e., the likelihood of change is low or predictable. Since a domain may be highly specialized, there is no general engineering methodology which could produce satisfactory results for all imaginable domains. Thus, domain-specific languages are typically used to describe domains [TMC99, vDKV00, MHS05].

In domain engineering, the commonalities of programs in a domain are detected and defined explicitly. Variability is understood as “the ability to change or customize a system" [vGBS01]. Furthermore, the variability characterizes differences in software products and is represented using the concept of a variation point which "identifies one or more locations at which the variation will occur" [JGJ97]. Variability and variation points are captured at different levels of abstraction, e.g., at requirements, architecture, or implementation level [BKPS04]. The exact description of where a variation point is located or how variability is finally implemented depends on the chosen method or description language. For example, variability can be represented at the requirements level as customizable use case scenarios, at the architecture level as optional components, and at the implementation level using parameterizable build files (see Sect. 4.3.3 for details). In all cases, there is a limited and predefined choice of options whose exact values are bound during application generation. Detailed discussions on the notion of variability can be found in [vGBS01, Bec04].

More details on the handling of commonalities and variabilities are presented in the following. The overall categorization of some techniques may differ from some presentations in the literature, since a more general perspective should be provided. The examination begins with domain analysis which comprises domain requirements analysis and scoping. It continues with domain design and illustrates feature models and production plans. Thereafter, domain implementation and testing techniques are discussed.

4.3.1 Domain Analysis

Domain analysis is a process performed at the beginning of domain engineering to systematically analyze the problem domain and to structure the knowledge in a way that it is useful for the other phases of the development process [Ara94]. Although domain analysis has initially been developed for the context of component-based development [Nei80], it has been adapted for software product line engineering as well. The most important sub-processes in domain analysis are domain requirements engineering and domain scoping.

Domain Requirements Engineering

In traditional requirements engineering, the following activities are performed for single systems [Poh97]: requirements elicitation (understand customer needs and constraints), documentation in a structured form, negotiation (achieve consensus with stakeholders), verification and validation (ensure that requirements are clear, complete, correct with respect to internal consistency and external user validation, and understandable), and management (requirements are continuously kept consistent and up-to-date). In domain engineering, it is necessary to capture additionally the commonality and variability of software products in a product line. For example, this can be done with the following approaches [PBvdL05]:

- **Application-Requirements Matrix**
  The Application-Requirements-Matrix is structured as follows: it contains in the column heading all applications (i.e., software products) and in the row heading all application requirements. At the intersection of a row and a column, it is marked if a particular feature is mandatory for a particular application. Requirements which have the 'mandatory' mark in every cell of one row are candidates to be common requirements, whereas the others are variable requirements.
- **Priority-based analysis**
  Another way to obtain common and optional requirements is a priority-based analysis which lets stakeholders rate the requirements and define a priority for them. If a significant number of customers rate a requirement as important or indispensable, then it should be considered to categorize it as a common requirement for all products in the product line. Details on how to organize such surveys are given in [PBvdL05].
- **Checklist-based analysis**
  The checklist-based analysis identifies common requirements using checklists which may cover for example different categories of legal requirements or organizational standards. The analysis of such lists may reveal common requirements right away.

During the requirements engineering process for software product lines, it is conceivable to use all of these approaches. A feature in a product may be created due to one or a set of several requirements captured in this phase.

Scoping

Generally speaking, scoping is performed to answer the question of what is in the product line and what not. There are, however, several different areas in which boundaries can be defined [BKPS04]:

- **Product portfolio scoping**
  This area aims to identify which products should become a part of the product line, not only under the consideration of marketing aspects, but also based on the possibility to exploit commonalities. For software product lines, the products in the portfolio must have technical commonalities.
- **Domain scoping**
  In this area, the relevant domains for the product portfolio have to be identified along with their boundaries. In addition, the general description of areas of functionality is limited to the functionality needed in the software product line. The resulting domain descriptions are analyzed to identify potentially reusable parts (which are often modeled in later stages with feature models, see Sect. 4.3.2).

- **Asset scoping**
  This area focuses – in contrast to the other two – on how to realize the needed functionality. For this, reusable components (the "core assets", see Def. 4.1) offering different aspects of the functionality are identified in existing repositories or specified for an own development.

In principle, product portfolio scoping and asset scoping specify the "problem space" while asset scoping specifies the "solution space" of the software product line. In the literature, more specific scoping techniques have been developed. As an example, the main ideas of FAST and PuLSE-Eco are briefly described.

In the Family-Oriented Abstraction, Specification, Translation (FAST) approach used at AT&T and Lucent Technologies, commonality analysis firstly defines a standardized terminology for the domain [WL99]. Then, commonalities and variabilities are stated as assumptions. Commonality is understood as an invariant, and in particular as "assumptions that are true for all family members" [WL99]. Finally, parameters of variation and binding times are defined for each variability.

Within the Product Line Software Engineering (PuLSE) approach developed at Fraunhofer IESE, the PuLSE-Eco component incorporates economic aspects and connects the product line scope to business objectives. It is similar to the Goal-Question-Metric (GQM) approach which is a measurement mechanism that specifies goals on a conceptual level, concrete questions to achieve the goals on an operational level, and metrics for measurement on a quantitative level [BCR02]. PuLSE-Eco performs a step-wise scoping. Product line mapping captures the functional features of products and identifies the relevant domains. A product map is used to show which functionality has to be incorporated into which product. Then, domain potential analysis evaluates and identifies the areas with the highest potential for reusability using different evaluation criteria (e.g., for stability of domain, market potential, existing artifacts). Finally, reuse infrastructure scoping defines which reusable components are to be developed, along with quantitative, context-dependent evaluation functions that assess the benefit of introducing a feature into the reuse infrastructure. This approach has been evaluated in several practical scenarios. More details can be found in [BFK$^+$99, DS99, Sch03b, Sch04]. Other issues related to the implementation of scoping are discussed in [SG00].

As in general, it is also often difficult for the presented techniques to find the right balance between informal descriptions and too many formalized details. In this respect, it is possible that the described assumptions in the commonality analysis of FAST are not precise enough. On the other hand, the meticulous definitions employed in reuse infrastructure scoping of PuLSE are sometimes left out, as pointed out in [Sch04].

4.3.2 Domain Design

In the traditional design phase, abstractions and models are created for single systems (e.g., class UML diagrams, ER-diagrams, etc.). In the context of domain design for product lines, feature models are additionally created to specify all available features for all products in a product line, and constraints upon their selection. Thereafter, features of the feature models are connected to traditional design model elements (e.g., classes in UML) to indicate how they will be realized. Another typical output of domain design is a production plan which develops the details of how core assets are assembled to create products which have a subset of features of the overall feature model. The current practices related to feature models and production plans are presented in turn.

Feature Modeling

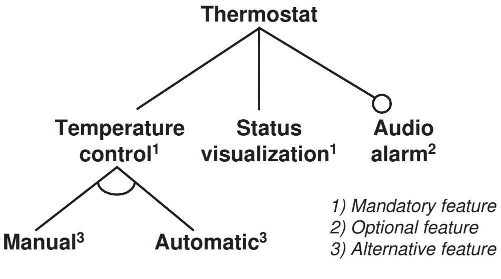
Throughout the development of the area of software product lines, feature modeling has remained a widely used technique to represent the commonality and variability of product variants on a feature level in an implementation-independent way. Feature models describe the available configuration space with all available options and constraints that are considered relevant. Feature models are an alternative to enumerating all valid combinations of features for every possible application, which would be inappropriate for large systems. Later in application engineering, the selected features for a product can be regarded as an instantiation of the product line feature model.

Although feature modeling is sometimes presented in the literature as a part of domain analysis, it is understood in this thesis that it is actually settled somewhere between analysis and design, as it transfers the requirements to a design for the product line. The resulting feature models are in principle a high-level design specification, so that it seems more suitable to discuss the related approaches here, and not in domain analysis. This view is also supported by some approaches which actually integrate feature models in [Gom05] or along [PBvdL05] traditional design models.

One well-known method for feature modeling is the Feature Oriented Domain Analysis (FODA), whose initial approach is used here to exemplify the typical activities of feature modeling in general [KCHP90]. FODA begins with a context analysis phase (which basically scopes the domain, see Sect. 4.3.1), followed by a domain modeling and an architecture modeling phase. The domain modeling phase which introduced feature models is described here in more detail. In FODA, features are understood to be characteristics visible to the end-user, which are further categorized into functional features (i.e., functions of the application seen by the user), operational features (related to the operation of an application from the user's perspective), and presentation features (related to the presentation of information to users). All features are represented in a feature model which consists of [KCHP90]:

1. **A feature diagram**
   a) The diagram graphically depicts a hierarchy of features, and has a distinguished root.
   b) Nodes other than the root represent features which can be mandatory, optional (drawn with a circle above the feature name), or alternative (drawn as children of the same parent feature, with an arc intersecting the connecting lines).
2. **Composition rules** may additionally express dependencies between features: mutual dependency ("requires") or mutual exclusion ("mutex-with").
3. **A record of trade-offs, rationales, justifications.**
4. **A record of system features.**

The feature diagram presents for the scoped domain all relevant features which can be eventually included into a product variant. In the diagram, there is no notational distinction between functional, operational, or presentation features. Composition rules are additional constraints limiting the choice of features. The record of trade-offs, rationales, and justifications offers guidance during the selection of features. The system feature catalogue is used to record which features are used in which systems with which values – it has therefore similarities with a product map (see Sect. 4.3.1). Each feature must have a distinct name that is also included in a domain terminology dictionary which is used throughout the modeling phase, and which describes the meaning of features. Finally, the validity of a feature model, i.e., whether it captures all relevant features and feature combinations, has to be verified by domain experts.



Rationale:
Manual has lower price
Composition rule:
Audio alarm requires temperature control = automatic

Fig. 4.3. An example for a FODA feature model.

The FODA diagram does not have precisely specified semantics. Therefore, the meaning of such a diagram will be explained using the example shown in

Fig. 4.3. Composite features (e.g., “temperature control”) may consist of several other features, while atomic features are not subdivided further (e.g., “manual”). A line in the diagram models the “requires” relationship between the possible features of the software for a thermostat, and in addition every feature has an imaginary flag (not depicted) to mark if it is chosen for a product or not. If a parent feature is not chosen in an instance, then all its children cannot be chosen. The root “thermostat” is, by definition, chosen for every product configuration. Then, the mandatory features “temperature control” and “status visualization” have to be implemented in every product. The feature “audio alarm” is optional, i.e, it is included only if desired by a customer. The alternative features “manual” and “automatic” are understood to be specializations of the parent “temperature control”; however, FODA allows only at most one of the specializations to be chosen. An additional composition rule specifies that when the optional feature “audio alarm” is chosen, then the temperature control must be “automatic”. Without such interconnections from composition rules, the feature diagram is basically a tree, however, with the introduction of such interconnections the diagram looses this property and becomes a graph. Finally, a rationale provides additional guidance, e.g., the choice of “manual” leads to a lower price for the product. The feature model is instantiated during application design, and examples for possible instances of Fig. 4.3 are given in Sect. 4.4.2.

The FODA notation has initiated the development of a multitude of other methods and notations for feature models. From a conceptual point of view, there are two main classes of approaches. In the first class, existing methods or notations are extended to incorporate variability descriptions (e.g., extensions to UML [Gom05]). In the second class, a separate orthogonal layer with a variability model is created, which contains variability descriptions that are independent of existing notations [PBvdL05]. From this layer, the parts that represent variation points can contain references to other underlying notations (e.g., a reference to a corresponding use case in a UML use case diagram or to particular calls in a UML sequence diagram). Both approaches have advantages and disadvantages. Extending existing methods accelerates learning, but the variable parts may be distributed across several models. A separate layer with a general variability model has the advantage that existing notations can be used as usual and that variability descriptions are separated. However, a disadvantage is that the model must be a quite general metalanguage which should be usable with several different types of underlying models, and a language with the ability to deal with all subtleties of all possible underlying models seems elusive. An appropriate choice might therefore depend on the particular project and domain.

There are currently many approaches for feature modeling which extend existing notations. As an example, Fig. 4.4 shows a comparison between the FODA notation (notation 1), a notation with additional constructs (notation 2) proposed in [CE00], and a cardinality-based notation (notation 3) presented in [CHE04]. All notations can express the mandatory, optional, and

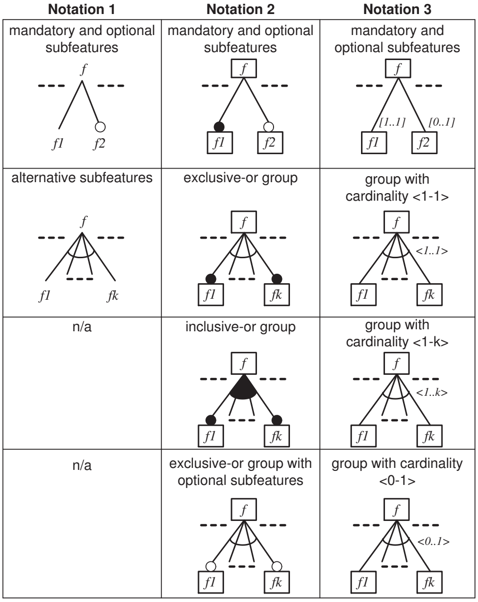

Notation 1
Notation 2
Notation 3
mandatory and optional subfeatures
f
f1
f2
mandatory and optional subfeatures
f
f1
f2
mandatory and optional subfeatures
f
[1..1]
[0..1]
f1
f2
alternative subfeatures
f
f1
fk
exclusive-or group
f
f1
fk
group with cardinality <1-1>
f
<1..1>
f1
fk
n/a
inclusive-or group
f
f1
fk
group with cardinality <1-k>
f
<1..k>
f1
fk
n/a
exclusive-or group with optional subfeatures
f
f1
fk
group with cardinality <0-1>
f
<0..1>
f1
fk

Fig. 4.4. A comparison of feature modeling notations (based on [CHE04]). Notation 1: FODA [KCHP90]; Notation 2: [CE00]; Notation 3: cardinality-based [CHE04].

alternative subfeatures of FODA. Notation 2 adds a construct to semantically denote groups consisting of one or more subfeatures, and has a constraint on the group – a feature can thus have several groups of subfeatures with different constraints on each group. From an inclusive-or group, any non-empty subset of features can be chosen if the parent feature is chosen. In the exclusive-or group with optional subfeatures either none, or at most one subfeature can be chosen. Notation 3 is basically an extension of notation 2 which explicitly shows user-defined cardinalities related to the constraints. Other extensions to FODA have been provided in the Feature Oriented Reuse Method (FORM) which is directed more towards object-orientation and adds a marketing and product plan [KLD02]. Another feature modeling notation has been developed in FeatuRSEB [GFd98] which extends the Reuse-driven Software Engineering Business (RSEB) approach [JGJ97]. FeatuRSEB is a use case driven approach where features are related to use cases or parts of use cases. Compared to FODA, FeatuRSEB has another notation for alternative features and distinguishes between variation point features (where a choice can be made) and variant features (the choices that can be selected). A description of FODA models in a textual way is offered in [vDdJK02], and [Bat05] shows the connection between feature models, grammars, and propositional formulas. Approaches with ontologies are shown in [dAFGD02, Ate04, CKK06]. The usage of XML for the description of a variability model is proposed in [Bec04]. Other approaches like Organizational Domain Modeling (ODM) [Sim95] or

Domain-Specific Software Architectures (DSSA), [TTC95] are discussed in [CE00, CHE04]. The usage of decision trees to select features has been investigated in [TGTG05], but becomes clumsy and impractical when the size of such trees grows.

Some points of the presented feature models which seem to have become common practice have to be criticized. One problem is that many of the notations are not defined precisely enough, e.g., using formal methods. This can lead to misunderstandings when one has to analyze what they exactly describe. It has also been shown that many extensions of FODA do not add any further expressiveness [BHST04]. Another problem is that some notations allow different ways to express the same issues (e.g., [CE00]) which may also lead to misunderstandings and complicated transformations when two given models have to be analyzed for equivalence. Furthermore, the notations often do not have a clearly-defined grammar and do not prohibit the creation of invalid models (cf. [vdML04] for examples). Another critical problem is that although feature models are presumed to have a tree-like form, the insertion of additional relationships between nodes actually lead to a graph. As such models become larger in practical applications, they also become more difficult to understand due to their interconnections, which degrades their usefulness for abstraction. Although in an automated analysis could be used in such cases [BBRC06], a feature modeling technique would be more helpful if the formalism itself is constructed in such a way that the discussed problems are not allowed to happen.

Production Plans

A production plan is the “prescribed way” of the product production mentioned in the definition of software product lines (cf. Def. 4.1), and is itself regarded as a core asset. The production plan is generally understood as “a description of how core assets are to be used to develop a product in a product line”, and is currently often specified only as a textual description covering the following points [CM02]:

1. Introduction
2. Strategic view of product development
3. Overview of available core assets
4. Detailed production process
5. Tailoring production plan to product-specific production plan
6. Management information

In the introductory part, there are descriptions of the production context, the audience, and the needed qualifications. Secondly, the strategic view makes the assumptions and qualities explicit. Furthermore, it is specified which products are possible based on available assets, and a production strategy defines (as a high-level statement) how the production goals should be met [McG04].

Next, the overview of available core assets describes their inputs, dependencies, and variations. Some authors assume that every core asset has an attached process related to its construction [CN02], and this description is also included in the production plan. Next, the detailed production process contains the technical details for construction. Despite the complexity, many production plans are currently simple textual descriptions [DTA05]. Alternatively, build scripts are used to implement the details of the production process (see Sect. 4.3.3). Next, product-specific descriptions are added. Finally, the part on management information contains organizational information, e.g., bill of materials, resources, team assignments, schedules, or metrics.

Although production plans are an essential part for the success of the product line approach, there is little research on the technical details, which goes beyond build scripts or how the textual descriptions should be structured. Sometimes the build scripts alone are understood to be the production plan. Overall, no specific modeling techniques were proposed which may simplify the modeling, analysis, verification, validation, or reuse of build processes for product lines.

4.3.3 Domain Implementation

In domain implementation, all the core assets are implemented which can be used to realize the previously defined features. In some approaches it is assumed that for every core asset there is an attached process which describes how to implement the asset (see also [CN02] and Sect. 4.3.2). In particular, the core assets which are of type "software component" are implemented at this stage. In addition, configuration and parameter passing mechanisms are included.

Parametrization can be used as a means to create variants of components with similar behavior. In principle, there are several binding mechanisms which can be used at different times of the implementation process, like for example [AG01, PBvdL05]:

- **before compilation**: code generation, aspect-oriented programming (AOP), model-driven architecture (MDA)
- **at compile-time**: pre-compiler macros
- **at link-time**: build scripts with parameters
- **at load-time**: configuration files
- **at run-time**: a system registry
- **at post-run-time**: binary patches

The binding mechanisms implement the variability of the product line in a predefined and controlled way, and parameters represent variation points. Before compilation, code generators can be used to generate code which has incorporated the common parts and, based on the value of parameters, the code pieces that differ in every product line member [CE00]. AOP can be used to insert predefined pieces of code (aspects) at specified program locations (join points) which can be in the existing code or in the control flow of the program (see [FECA04] for details). The MDA approach attempts to transform general, implementation-independent models into platform-specific models, which are then transformed into code [KWB03].

In the approach with macros, it can be specified at compile-time which parts of code will be compiled and which not (e.g., IFDEF statements in C++), according to macro parameters defined in the macro code.

At link-time, build scripts (e.g., using MAKE [Mec04] or ANT [The07]) can be parameterized to produce applications from different sets of base artifacts.

Another possibility is to implement components in such a way that they check at load-time a specified configuration file and adapt their behavior accordingly.

If a central registry is available in the target environment to store values at run-time, it can be queried by components at run-time to adapt their behavior or find other components.

Finally, the behavior of components can be also altered at post-run-time (i.e., update time) [AG01] through binary replacement of the whole component of parts of it. A more detailed classification of variability realization techniques is given in [SvGB05].

The difference of domain implementation from the implementation of single systems is that after the development of core assets, there is no executable application yet in this phase. Applications are created later in application engineering with the core assets produced here. Another difference is that the implemented artifacts incorporate the variability defined in the feature models, using the mechanisms described above. These differences require also special testing approaches which are described next.

4.3.4 Domain Testing

In general, a test is an "activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component" [IEE90]. Domain testing differs from traditional application testing, as it poses the problem that after domain implementation there exist only core assets (e.g., code components), but not yet a complete executable application which can be tested – this is constructed later in application engineering. However, it is desirable to test core assets at this point, especially those representing common parts used in all systems, in order to be able to make corrections as early as possible. A particular difficulty is that the available core assets could be combined in different ways to obtain a concrete system, resulting in a large number of possible configurations for a concrete system.

Domain testing has to deal with loosely coupled components that were produced in domain implementation, and try to discover defects (i.e., if the specified behavior is different from the implemented behavior). Test artifacts produced in domain testing, e.g., test cases (specifying initial conditions, inputs, expected results), test plans (describing what and how to test), and test reports, are intended to be reusable artifacts which can be used in this phase, but also later in application testing. Test artifacts may also incorporate variability. For example, a test case generator may produce random text according to predefined parameters.

Like for single systems, domain testing must occur at different levels of granularity. At the lowest level, domain unit tests validate modular parts of code against the specified behavior and are in most cases feasible. A domain integration test which validates several interacting components is typically infeasible, since not all components have to be integrated in an application, and there might be many possible configurations. The creation of test mockups (or stubs) that "simulate" the behavior of unfinished components is also problematic, because they require themselves effort to be created, are not a fully equivalent substitute for the original component, and might be themselves the source of errors. A domain system test in which a system is tested that represents the whole domain is typically infeasible as well.

Various strategies are identified in the literature to cope with the aforementioned problems [PBvdL05]:

- **Brute Force Strategy (BFS)**
  The BFS aims to perform tests at all levels for all possible application configurations. This strategy is in practice usually infeasible since the number of applications with different configurations can grow exponentially [TTK04].
- **Pure Application Strategy (PAS)**
  The PAS does not do any testing in domain engineering, and postpones all testing to application engineering, when an application is created. However, an early validation would be missed this way and there would be no possibility to reuse test cases. The repetitive development of test cases from scratch is a major drawback, especially because there might be behavior that is common to all applications in the product line.
- **Sample Application Strategy (SAS)**
  The SAS intends to create in domain testing a sample application with a typical configuration. The common components are tested in this context, enabling an early validation of common features. The generated test cases can also be reused later (possibly with minor adaptations) in application testing. One drawback of SAS is the overhead to create a sample application.
- **Commonality and Reuse Strategy (CRS)**
  The CRS distributes the testing activities in another way: in domain testing, test cases are defined for the common and variable parts of an application. Common parts are tested with the appropriate test cases as far

as possible. Later, all predefined test cases are reused in application testing for a chosen system configuration to test the variable parts and once again the common parts to see if they work as intended. In [PBvdL05] it is suggested to use in practice a combination of SAS and CRS.

Beyond these high-level strategies, there are also more detailed approaches for certain contexts of domain testing. For example, the ScenTED approach [RRKP06] derives test cases based on a test model that is itself derived from the requirements. A strategy for regression testing in product lines is sketched in [McG01]. Other approaches are sketched in [KN06, KD06], and in a survey in [TTK04] which observes that many testing concepts for product lines are in their infancy, especially those around the code level. Further research is needed, especially with respect to the concrete division of tasks between domain testing and application testing, since they can be difficult to split up in practice.

4.4 Application Engineering

Application engineering focuses on the creation of a single product, using the prepared models and artifacts from domain engineering [PBvdL05]:

Definition 4.3 (Application Engineering).
Application engineering is the process of software product line engineering in which the applications of the product line are built by reusing domain artifacts and exploiting the product line variability.

Every product within the scope of the product line can be created in application engineering by instantiating the models and reusing the artifacts of domain engineering [vGBS01]. The parameterizable core assets are configured with concrete parameters. Then, they are assembled to realize the needed features. However, many approaches allow minor product-specific additions if they are in line with the overall product line specifications defined in domain engineering. If domain engineering was done right, the effort in application engineering should be much lower than in single system development. Some relevant details of application analysis, application design, application implementation, and application testing are presented next.

4.4.1 Application Analysis

The application analysis process is mainly focused on the requirements analysis for a concrete application. One difference from requirements analysis in single systems is that here a balance has to be found between the stakeholder requirements and the product line requirements defined in domain engineering for all applications in the product line. The idea of the product line as well as its scope have to be communicated to the customer in order to negotiate which requirements are really needed, and if they can be realized with the product line. Of course, there might be tradeoffs for customers when they do not receive 100% of the features they want, but if they can live with, say, 90% of the features which can all be delivered with the product line approach for a fraction of the cost of a single system, this approach becomes attractive. Furthermore, using the product line solution, customers can expect a higher quality and better stability of an application, since the underlying core assets must themselves be of higher quality, as they are reused in several systems. Nevertheless, required additions that are not present in the product line have to be recorded and evaluated carefully to make sure that they fit into the existing context.

Other typical activities in application analysis are the documentation and the traceability of application requirements, especially with respect to the connection to domain requirements. Furthermore, the parameter values for variation points are defined, and they will remain fixed from now on. Application analysis influences also domain analysis. For example, if in the requirements documents of different applications it is detected that many customers demand a certain functionality that is not already in the product line, this feedback must be communicated to domain engineering to help to improve the overall design.

More details on application requirements engineering can be found in [PBvdL05]. An approach of how to communicate the product line requirements to customers is presented in [HP03]. The approach in [KS00] describes how to capture decisions and priorities for requirements.

4.4.2 Application Design

During application design, the architecture and the corresponding models of an application are created, using the preparatory work of domain design. Based on the results of application analysis, one of the main activities of application design is to choose a valid feature configuration for an application. This configuration is in principle a valid instance of the feature model developed in domain design (discussed in Sect. 4.3.2). As an example for such an instantiation, valid feature sets derived from the feature model in Fig. 4.3 are shown for product variants for the software of a thermostat. They are enumerated below (the root node is omitted and composite features contain subfeatures in parentheses):

In addition to the generation of a consistent feature configuration, other models possibly defined as a reference in domain design are specialized here for the particular application, and variation points are bound to concrete values (e.g., “temperature control” is bound to “manual” in $Variant_1$). The selected features influence the concrete architecture of the application. The generic production plan may also be adapted to account for the peculiarities of the application, like for example individual adaptations specified in application analysis.

$$\begin{aligned}
Variant_1 &= \{\textit{temperature control (manual), status visualization}\}\\
Variant_2 &= \{\textit{temperature control (automatic), status visualization}\}\\
Variant_3 &= \{\textit{temperature control (automatic), status visualization,}\\
&\qquad \textit{audio alarm}\}
\end{aligned}$$

Fig. 4.5. An example for instances of the feature model in Fig. 4.3.

A difference between application design in the product line context and traditional single system design is that the application architecture in the product line context is based on the results of domain design and may also give feedback to domain design (e.g., useful changes to the reference architecture). In addition, reuse in application design occurs on an architectural level, and it is ensured – due to the preparatory work in domain design – that the refinements of the general architecture of domain engineering will lead to a functioning system.

However, there are still many sources for errors as well as open research questions, especially with respect to empirical validations how well the application design can be derived from the domain design. Although some case studies describe positive results [ADH$^+$00, CN02] (sometimes from a general perspective) there is also the awareness that in the presence of tens of thousands of variation points there can be a large number of human errors during product derivation, due to complexity of the models (see case study in [DSB04]). The latter results support the criticism put forward in Sect. 4.3.2 that most of the proposed feature models are difficult to use at larger scale, and that they may be ambiguous due to a lack of formal precision or disadvantageous model constructs. These shortcomings must be remedied if application design should work as intended.

4.4.3 Application Implementation

Due to the preparatory work done in domain engineering and in the previous phases of application analysis and application design, the number of activities in application implementation is reduced. As a difference from traditional systems implementation, the focus of this implementation process is often shifted from a detailed development to a configuration and assembly of software components [CN02]. Based on the specific application architecture, the reusable components created in domain implementation are selected and configured with the specified parameters so that they exhibit the desired behavior. If the work in the previous phases was done right, the effort to create individual components for the specific application (which are not part of the product line architecture and on which a customer might have insisted) should be minimal. The application architecture specifies how the components interact, and provides the means to integrate them into a functioning system. The details on how to compile and build the necessary files can be taken from the production plan. More details on application implementation techniques are discussed in [CE00, AG01].

4.4.4 Application Testing

As already discussed in Sect. 4.3.4, some activities of the testing process in product lines are done in domain engineering and some in application engineering. Application testing can of course test the final application with traditional testing techniques (see [AM05] for a comprehensive overview).

Application testing can be done on different levels: unit testing (validating the single and newly created components), integration testing (checking component interaction), system testing (for whole application). However, prepared test artifacts from domain engineering, like for example test cases, test plans, or test reports, can be reused and configured for the particular application. Depending on the chosen strategy of domain testing (e.g., Sample Application Strategy or Commonality and Reuse Strategy), some parts of the application may be already tested, which can speed up application testing.

The common parts need to be tested in a commonality test with reusable test cases to see if they still work in the current application context. For variability testing, there are two further types of tests: the variant absence test and the application dependency test [PBvdL05]. The variant absence test makes sure that no functionality was included that was not defined in the configuration of the particular application. An application dependency test checks that no constraint defined in the domain models is violated in the generated application. These types of test are typical of product lines and usually not needed in traditional single-system development [TTK04]. Finally, application-specific tests may be performed.

4.5 Family Engineering

In the sequel, "family engineering" will be used in a sense that is related to the product line as as whole, with activities which may crosscut domain or application engineering[2]. Family engineering is understood here as a subprocess in its own right which subsumes, based upon existing approaches, under one roof economic aspects, organizational aspects, and product line evolution aspects. Such an integrated view of these issues as a subprocess in its own right is often neglected in the literature or roughly reduced to "management" [CN02].

[2] Remark: there are approaches which have a different understanding of family and domain engineering. For example in [Mut03], a domain is understood to potentially characterize an infinite number of applications of a particular type; family engineering belongs there to domain engineering, and a family is only a subset of systems of the domain.

4.5.1 Economic Aspects

There are currently different approaches to assess if the investment in a software product line is worthwhile or not. In this context, several methods known from investment theory can be applied.

Starting on a rudimentary level, the basic principles are illustrated in Fig. 4.6. The figure compares the cost of development with domain engineering (in a product line approach) with that of the traditional single-system development without domain engineering[3]. The cost of developing one system in the traditional way is denoted as C_T, so that the development of N product line members yields the total cost $N * C_T$ (line A in the figure). With domain engineering, at first the investment I has to be made for example for the domain engineering itself (cf. Sect. 4.3) and the creation of a core asset base. This investment can also be risky, since it can delay the delivery of the first product in a product line. However, later the creation of additional product line applications is faster than in single system engineering. Assuming that C_F is the cost for producing one family member, the cost of producing N family members is $I + N * C_F$ (line B in the Figure). If domain engineering was successful, then $C_F < C_T$. This is the case in Fig. 4.6, i.e., the slope of line B is lower than that of line A; an intersection (=payback point) of both lines exists.

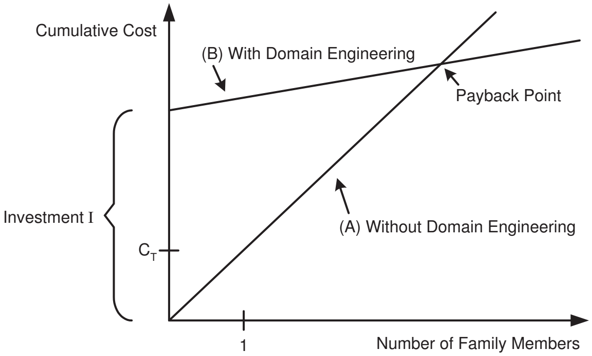


Fig. 4.6. A situation with successful domain engineering (based on [WL99]); C_T ist the cost of developing one system in the traditional way.

In the literature, there are different estimates of when the investment in the product line approach pays off, i.e., of how many systems need to be developed so that the development with product lines is more efficient than

[3] In [WL99] the shape of the curves is assumed to be linear. Although the number of systems and the associated costs are usually discrete values, the lines are drawn continuously for illustration.

with the single-system approach. For example, [WL99, CN02] estimate for their presented context that between 2 and 4 systems are needed. Furthermore, [WL99] report a decrease of development time of factors ranging between 3 and 5. In another case study [ADH$^+$00], the initial investment I is reported to be less than the cost of three new family members, and the reduction of the marginal cost with domain engineering (C_F) is reported to be on the order of 4 to 1. Overall, these numbers have been obtained in the particular contexts, so that one cannot say for sure to what degree they can be generalized.

A more detailed cost model taking into account the peculiarities of product lines is proposed in [BCM$^+$04]. The cost to establish a product line with N products is quantified as:

$$C_F := C_{org} + C_{cab} + \sum_{i=1}^{N} (C_{unique}(p_i) + C_{reuse}(p_i))$$

where $C_{org}, C_{cab}, C_{unique}, C_{reuse}$ are cost functions. However, they are defined only in a general way. C_{org} are costs to adapt the organization to a product line approach, train staff, etc; C_{cab} is the cost for the development of a core asset base as well as the cost of the associated commonality and variability analysis; C_{unique} is the cost to develop parts that can be unique to a product p_i in the product line, and it is assumed that they are usually small; C_{reuse} are the costs that are incurred for a product p_i during reuse of existing core assets, e.g., for searching, checking, or integration. The authors also propose different variations of the formula depending on the usage scenario. However, it is not described how to exactly estimate the different cost components. It is proposed to use case studies or standard reuse metrics [Pou96] for C_{cab} , other known models like COCOMO [Boe81] for C_{unique}, or historical data of the company. The latter can be a problem especially for young companies. Other problems of this cost model are that it is not obvious how other regularly incurring costs are considered which can result from evolution, e.g., for continuous updating and maintenance of the core asset base.

In the presented methods, the mere concentration on the costs and the payback of the initial investment does not take into account that the money could have been invested somewhere else, possibly accruing interest over a period of time. Therefore, a “good” investment would not only earn its initial amount of money back, but additionally earn at least the interest that could have been earned for example by depositing the money on a bank account. To be able to compare cash in- and outflows of different time periods, their net present value has to be calculated for the current period ($t = 0$):

$$NPV = \sum_{t=0}^{n} \frac{C_{i_t} - C_{o_t}}{(1 + r)^t}$$

where C_{i_t}, C_{o_t} are the cash in- and outflows for a period t; r is the opportunity cost of capital, i.e., the interest that could have been earned in another similar investment; t is a discrete time period; n is the total number of periods. Using estimates for the cash flows for the development of a set of products with a product line and without a product line, the $NPVs$ can be calculated for both alternatives (that are assumed to be mutually exclusive) and the alternative with the higher, non-negative NPV should be chosen (see also [Wit96]). The general problems with this method, such as the assumption of perfect markets with equal interest rates for debit and credit, certainty of cash flows (and neglect of risk), focus only on tangible benefits, the determination of the discount rate r – to name just a few – are well-known and thoroughly discussed in the literature [BMM99].

Without unfolding here the well-known investment theory into too many details, it should be mentioned that other approaches can be used as well in the context of product lines. For example, decision trees can be used to model the possible decisions (e.g., make initial investment? produce repository?) and possible outcomes (e.g., the respective NPV) with their expected probabilities of occurrence. In contrast to the traditional NPV method, decision trees are able to model interventions depending on how the current situation develops (see also [FFF98] for software-related examples). It has also been recognized that such decisions resemble "real" options (as a counterpart to financial options, see [BMM99]), and that option pricing theory [BS73] can be applied to estimate the value of options in reuse investment. The main rationale is that the investment in a reuse platform, e.g., domain engineering and a core asset base, will not only produce directly measurable cash flows, but also the opportunity (=option) to react more quickly on the market, or even enter new markets (see [BS73, FFF98] for more details). From a different perspective, the core asset base and the possible products in the product line can be regarded as a portfolio which helps to diversify risk [Wit96]. A related method to reduce risk is an incremental introduction of the product line approach [SV02].

It is obvious that each of the methods has its advantages and disadvantages. Although the simpler methods have some inaccuracies, they are useful when a limited number of facets are of interest during the evaluation of an investment in product lines. By contrast, the more complex methods incorporate additional information, but often introduce additional parameters which require the collection of additional data. However, there might be still situations in which not all possible developments can be quantified. For example, a product line and its underlying platform may eventually evolve to an industry-wide standard – a situation which would give a company an important strategic advantage and an opportunity to influence competitors in unforeseen ways [MS98]. For product lines, other specific contextual factors can play a role for cost determination, which would require further adaptations of the investment analysis methods. Adaptations and additions can be, for example, the rate how often core assets have to be updated in order to keep all products in the product line competitive, or the maintenance effort for the production plan [Coh03], or quality cost [IBK$^+$06]. From a marketing perspective, the inclusion of additional products may lead to cannibalization effects where the introduction of a new product might take away sales from other products in the product line.

4.5.2 Organizational Aspects

The discussion so far is now complemented with a brief presentation of some proposed approaches on how to organize people and business units for the development with software product lines. In [Bos01] various forms of a basic organization are proposed, which all have different advantages and disadvantages, and which are suited for different situations:

- **The development department model**
  This model does not impose a certain organizational structure for product lines, and every person in a development department can be assigned tasks from domain or application engineering. The department is the only organizational unit dealing with the product line. This approach is applicable in small organizations and facilitates communication between developers. As a disadvantage, this form of organization is not scalable.
- **The business units model**
  This model has business units as organizational entities dedicated to product line development. Each unit develops one or a few products in the product line. Domain engineering and the core asset base are shared between the business units. Business units can create new assets or extend existing ones and make them available for the other units. If the coordination needed for the update of the shared assets is not controlled, then there is a danger that the performed extensions are not general enough. To deal with this, the responsibility to control the evolution of one asset can be assigned to one of the business units. However, it can be technically difficult for a unit to check whether an asset was only changed as agreed upon. As a compromise, the responsibility for an asset can be assigned to the business unit in relation to the products it produces, for example to the one which makes the most extensive use of an asset. The advantages of the overall approach are better scalability and exchange of core assets. As a disadvantage, there might be conflicts between the business units (especially if they are profit centers) so that there is no incentive to reuse assets from other units.
- **The domain engineering unit model**
  This model separates for larger organizations domain engineering and application engineering into one domain engineering unit and several product engineering units. The domain engineering unit creates the product line architecture and the core assets. In even larger organizations, the domain engineering unit can again be split up on the same level into a group responsible for the architecture, and a group that creates the core assets.

The product engineering units create products derived from the product line specifications and the requirements of customers. An advantage of this organizational form is that a product engineering unit has to communicate only with the domain engineering unit. One drawback is the difficulty to transfer the wishes of the customers to the domain engineering unit which also has to deal with possibly conflicting demands of other product engineering units.

- **The hierarchical domain engineering units model**
  This model targets larger organizations in which a hierarchy of domain engineering units is established for a better coordination. One leading domain engineering unit coordinates several specialized domain engineering units which focus on the development of a subset of core assets. This organizational form has advantages if a large number of products and a large number of staff are involved, probably also at different geographical locations. One disadvantage introduced by the hierarchy is the reduced ability to quickly react to market changes, compared for example with the development department model.

Beyond these proposals for a basic organization, additional secondary organization structures which superimpose the basic ones, as well as other influencing factors known from organization theory are discussed in the context of product lines in [PBvdL05]. A comparison of different process management approaches for product lines is made in [VK00].

4.5.3 Evolution

Another important aspect is the evolution and maintenance of the product line after the initial creation of models and core assets. To a certain degree, product line evolution can be influenced through proactive planning. However, there might also be unplanned developments which demand a reaction. Evolutionary development for product lines can be categorized into proactive and reactive evolution [BKPS04]:

- **Proactive evolution** is planned in advance.
  - **Platform expansion** can incrementally add new core assets or new parts in the architecture.
  - **Platform enhancement** incrementally improves existing artifacts or the architecture.
- **Reactive evolution** results from unforeseen changes in the environment, and may affect several or all products in the product line. Adaptive or perfective maintenance operations can be involved [Som04b], as there may be changes in the scope of the product line, manifested as changes in functional, non-functional, or domain requirements.

- **New functional requirements** demand new functionality.
- **New non-functional requirements** (e.g., reduction of execution time) can result for example from the existence of newly available hardware.
- **New domain requirements** can affect domain engineering, for example, due to new laws or legal regulations.

A successful management of the product line evolution needs metrics and feedback mechanisms to measure various aspects of the product line and the products contained therein. Traditional metrics from software development [Zus94] and reuse [Pou96] can be applied in this context. However, special metrics are currently researched for product lines (e.g, number of products, defect density, percent reuse of core assets, architectural conformance; see [ZC03] for details). A maturity model for product lines and related artifacts is proposed in [Bos02]. More details on product line evolution are given in the survey of [Pus02].

4.6 Deriving a Precise Definition

The exposition so far has illustrated that there are several approaches with sometimes differing views and understandings of product lines and the components they consist of. At a bottom line, a precise definition is needed [OPS06] because 1) the meaning of the terms as they are used in this thesis needs to be communicated (e.g., a special feature model that is referenced later); 2) different views and ideas are collected from different approaches into one definition; 3) it serves as a basis for the understanding and the analogies to be developed in the following Chapters. However, it is not the intention of this definition to express all existing views of all existing approaches.

Definition 4.4 (Software Product Line).
A software product line $SPL(F, FT, P, A, C)$ consists of

- ***A set** F **of features** which each have a name, a type, and an annotation (e.g., a description in natural language). That is, $f \in F : f = (name, type, annotation)$; $type \in \{common, optional, alternative\}$.*
  It is distinguished between the sets of
  - *common features $CF := \{f \in F \mid type(f) = common\}$ with $CF \neq \emptyset$*
  - *optional features $OF := \{f \in F \mid type(f) = optional\}$*
  - *alternative features $AF := \{f \in F \mid type(f) = alternative\}$*

- *The features F are organized in **a feature tree** FT with vertices (or nodes) V. Each node $v \in V$, except the root node $r \in V$, has a type and contents, i.e., $v = (type, contents)$ with node $type \in \{common, optional, alternative\}$. Via its contents, every node is linked to features in F, and* *each feature must occur in exactly one node. As an exception, r has no corresponding feature in F. The connection between a node v and features is as follows:*

- $$contents(v) = \begin{cases} f \in CF & \text{, if } type(v) = common; \\ f \in OF & \text{, if } type(v) = optional; \\ \left.\begin{array}{l} (Y, min, max) \text{ with} \\ Y \subseteq AF; min, max \in \mathbb{N}; \\ min \leq max \leq |Y| \end{array}\right\} & \text{, if } type(v) = alternative \end{cases}$$
- *if $type(v) = alternative$ then v must be a leaf in FT.*

- ***A set of software products*** *$P = \{p_1, \ldots, p_q\}$, and for $p \in P$: $p = (pid, FT_p, F_p)$ with*
  - *pid is a unique identifier for the software product p.*
  - *FT_p is a feature tree for the product p, which is called an instance of FT. This means that FT_p is a subtree of FT, with root node r that is by default included into FT_p[4];*
    - *a node v of type common in FT has to be included in FT_p if its immediate predecessor was included, i.e., for an included node, all immediate successors of type common have to be included in FT_p[5];*
    - *from a node v_a of type alternative in FT, not all alternative features must be chosen for FT_p. The corresponding node $v'_a = (type', contents')$ in FT_p has $type' = alternative$ and $contents'(v'_a) = X \subseteq Y$ with $min \leq |X| \leq max$, i.e., at least min and at most max alternative features have to be chosen from the set Y[6].*
  - *The set $F_p = CF_p \cup OF_p \cup AF_p$ denotes all common, optional, and alternative features in FT_p which are finally in the contents of the included nodes; these features will realize the functionality of the product p.*

- ***A set of core assets*** *$A := \{c_1, \ldots, c_r\}$ which are used to build a feature or a subset of features in FT. Furthermore, for $c \in A : c = (cid, content, annotation)$, which means that a core asset can conceptually consist of*
  - *a unique identifier cid;*

[4] From “subtree” it follows that if a node of FT other than r is included into FT_p, then its predecessor must also have been included.

[5] It follows that a node of type *optional* may be included in FT_p if its predecessor is included; therefore, this condition is not part of the definition.

[6] The notation of $contents'(v'_a)$ is simplified to avoid unnecessary overload. A formally more precise formulation would be $contents'(v'_a) = (X, n, n)$ with $|X| = n$ and $min \leq n \leq max$, since $contents(v)$ is a triple. However, min and max need not be transferred to the instance FT_p since the respective instance node is intended to contain only the subset X of selected features.

  - *some content which can be for example a document, code, a model, etc.;*
  - *some annotation related to the content, e.g., natural language descriptions, metadata, process specifications for its construction, etc.*

- ***A construction specification*** $C = (model, annotation, B)$ *which specifies how to create every possible product of the product line from core assets. In particular,*
  - *there is a model which describes the overall construction;*
  - *an annotation adds additional information, e.g., as natural language descriptions;*
  - B *is the built-from-relation* $B \subseteq P \times A$ *indicating which products are built using which core assets.*

As defined in Def. 4.4, a software product line SPL is understood to consist of a set of features F that are organized in a feature tree FT, a set of software products P that can be built using core assets A, and a construction specification C that defines how to built products from core assets. Three types of features are distinguished: common features, optional features, and alternative features. In the feature tree, each node is “linked” to features, and a feature must occur in exactly one node. The set of common features must not be empty, since without commonalities, the setup of a product line would hardly make sense. The types for features and nodes are used to impose constraints on the choice of features for particular products. The feature tree is used as an organization structure for features and as a means to model feature dependencies and constraints.

An instance of the feature tree is used to model valid feature configurations for concrete products. For the creation of such an instance, it is assumed that starting with the root (which is always included in every product) only those nodes from FT are chosen, which contain features that should be implemented in a product. For a specific product, the resulting instance FT_p is also a tree. The selection process of features in the nodes of the tree FT is influenced by already chosen parents and by the type of a node. An application engineer may be forced to select a node with a common feature if its parent node was selected; a node with an optional feature can be selected if needed, provided that its parent node was selected; if the parent of an alternative node was selected, then a certain minimum and maximum of alternative features have to be selected. The core assets in A have, in addition to a unique identifier, a content part and an annotation part. The content part can be for example a code component, an architecture specification, a design diagram, a test case, etc. The annotation part can be for example metadata, a natural language description with details on how to use the asset during the construction of a product, or even a more precise process specification for application engineering (similar to the proposal of [CN02]). The construction specification C is aimed at the construction of the product line as a whole, and contains a model specifying the overall construction procedure for every product, and an additional annotation which can be for example in natural language. Such an annotation can contain the non-model information of a production plan (e.g., the textual descriptions shown in Sect. 4.3.2). Finally, the built-from-relation B captures which core assets are used to build which product.

Example 4.1

Figure 4.7 is used to illustrate the introduced feature model and the creation of a valid feature configuration. It shows the tree FT as a possible excerpt for a product line of accounting systems, which can be configured to meet various needs with respect to the graphical user interface (GUI), law regulations, or reporting.

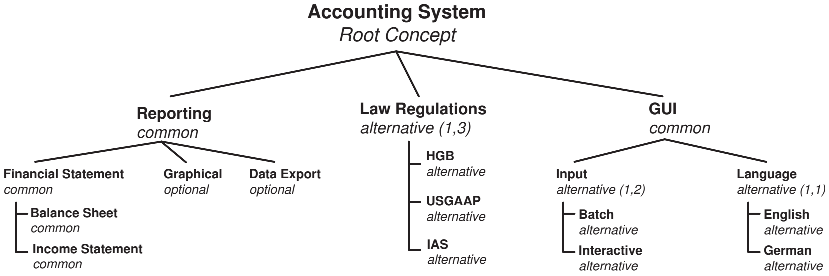


Fig. 4.7. Example for a feature tree FT for accounting systems.

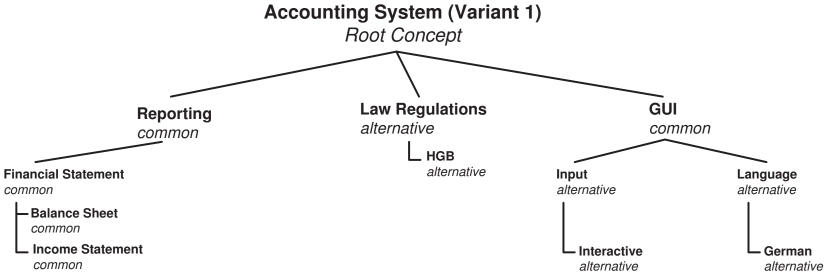


Fig. 4.8. Example for an instance $FT_{variant_1}$.

Here is a valid feature configuration for $variant_1$, a product for the low-cost segment of the German market (the root node is omitted and subfeatures of a feature are written in parentheses):

- {*reporting (financial statement (balance sheet, income statement)), law regulations (HGB), GUI (input (interactive), language(German))*}

The corresponding tree $FT_{variant_1}$, which is an instance of FT, is shown in Fig. 4.8. The construction specification will contain the needed details on how to build a system with these features from the existing core assets. □

4.7 Summary and Discussion

Contrary to an intuitive belief, software reuse is difficult to realize in practice – an insight that crystallizes out throughout the history of software engineering. The approach of software product lines is realistic enough not to believe in a too general and opportunistic (i.e., unplanned) reuse. Instead, it is believed that coordinated and planned restrictions will improve reuse. The approach of software product lines can be understood as a small niche in the reuse landscape, which makes extensive planning in advance, and which assumes that several similar systems will be produced. The main trade-offs are known from the beginning: a fixed number of valid configurations, a fixed set of features and possible implementations vs. improved reuse, lower costs, better quality, easier maintenance. This approach will only be applicable when the assumptions and the trade-offs are accepted. The most important points of software product lines, which are also of interest in the following Chapters, are:

- software product lines are a **special-purpose** reuse approach with specific assumptions;
- software product lines focus on **sets of similar software systems**;
- most **commonalities and variabilities** of product variants are known and specified in advance (e.g., as predefined feature models, core assets, construction rules);
- most effort is spent in domain engineering to **prepare reuse** for application engineering that mainly focuses on assembly of artifacts produced in domain engineering;
- the reuse **context** of artifacts is known and planned in advance.

In the research of software product lines, however, there are still many open questions. For example, the idea of feature models was quickly adopted by the community, but the concrete modeling techniques lack more precise semantics and tool support for more complex scenarios. The diversity of available feature modeling notations probably points to a need for different notations for different domains or to a need for standardization of notations. Testing in the context of product lines provides also opportunities for research, especially on how to connect domain and application testing. Further open questions are currently also encountered in the area of process management for software product lines, especially with respect to process modeling and process reuse.

Part II

Product Lines for Digital Information Products

5

The Product Lines for Digital Information Products (PLANT) Approach

This Chapter introduces the **Product Lines for Digital Information Products (PLANT)** approach that is targeted at the creation of variants of digital information products. The Chapter begins with the motivation and description of the goals of this approach, and presents thereafter typical shortcomings of existing approaches. The discussion of shortcomings is used to derive requirements for PLANT. This is followed by a macroscopic and general overview of the PLANT approach, which describes the main ideas such as the underlying strategy, assumptions, constraints, the process model, as well as a related tool. In addition, ideas of validation of the approach are discussed. The Chapter concludes with a summary and a discussion of the expected benefits.

5.1 Motivation and Goal

The discussion in Chapter 2 has revealed that digital information products generally play an important role in today's digital economy, and that substantial investments have been made in the creation of various types of content that constitute the core benefit of digital information products. In Sect. 2.3, the attention was drawn upon the need for an efficient creation of versions of digital information products. Due to the properties of digital information products – indestructibility, transmutability, reproducibility, and the typical cost structure – there is not only a technical need, but also an economic pressure to create variants of digital information products. Thus, one can work against saturation effects in markets, and address different customer segments. In addition, it was concluded that a structured approach is needed which helps to reduce the costs of production, for example, by taking advantage of possible commonalities within the content that is used in different digital information products.

The importance of content contained in digital information products has been illustrated in Chapter 3 for the e-learning domain as an example area where content is one of the critical parts to establish a competitive advantage.

Although progress has been made with reuse techniques in the area of Software Engineering, the e-learning area at this point shows that the reuse of content within digital information products is still immature. To some degree, the history seems to repeat itself: the reuse approaches for content within digital information products in e-learning seem to run into similar traps like the reuse approaches for code in the early days of Software Engineering, where it is meanwhile noticed that it "is far too simplistic to assume that components are simply selected from catalogs, thrown together, and magic happens" [SGM02]. Especially in e-learning it is evident that content reuse approaches should also address higher conceptual (i.e., strategic) levels, not only fine-grained technical levels.

This is a motivation for this thesis to transfer the experience and extend key concepts of reuse from Software Engineering, especially those of software product lines (presented in Chapter 4), to digital information products. As pointed out, the content contained in today's digital information products may consist of a mixture of text and software (see Sect. 2.2.5). Software Engineering techniques are therefore needed anyway, however, with adaptations to make them applicable in the specific context of digital information products. The details of these adaptations will be developed throughout the thesis.

As learned from the e-learning area, opportunistic reuse of digital artifacts within digital information products is inefficient and needs to be replaced by a structured, engineering-like reuse approach that takes also strategic aspects into account. These include planning reuse before artifacts are created as well as planning from the beginning what the context of reuse will be. Furthermore, assuming that the content found in variants of digital information products is similar to some degree, reuse has to keep an eye not only on individual products, but also on the sets of similar products. Only this way one can take advantage of synergy effects and reduce the costs to produce digital information products that have commonalities. The focus on sets of similar products is indeed realistic, as can be observed in e-learning; for example one can think of the digital content for an information systems course which is offered for different audiences (e.g., beginners, intermediates, experts), or with a different emphasis (e.g., database-centric, workflow-centric). Such content often possesses common parts and parts which are different, which can be explained by the typical creation procedure. It is conceivable to create the intermediate version first, and then delete some parts for the beginners or add some parts for the advanced course. Other strategic aspects also play a role for reuse, since such content is typically not developed in isolation, but has be to aligned with the strategy of a department, an institute, or a university as a whole. In many situations, an underlying strategy additionally regulates which content is covered by courses in a department, and how they need to be integrated into an overall curriculum.

In a different area, the example of Google Earth (see Sect. 2.2.2) shows that different customer groups may receive different information with different levels of additional detail. The same is true in a case of a digital newspaper which might offer a base package with news for the general public, and additional configurable categories of news for financial analysts, managers, or scientists. These observations and the sketched requirements additionally motivate the application and adaptation of the software product line concept to digital information products. Therefore, the goal of the approach to be developed is to reuse content within digital information products in a systematic way, and especially handle the commonalities between similar products in a systematic way. For a successful reuse, the approach must ensure that the combination of available content components is finally feasible and that all pieces will fit when they are assembled together.

5.2 Shortcomings of Existing Approaches

Although the reuse process of digital content seems to be immature at the moment, it has not attracted much attention, as in the early days of the World Wide Web the amount of digital content and the complexity of digital products were small enough to be handled just “by hand”. However, this has changed in recent years dramatically as digital information products have become more complex and ubiquitous, mainly propelled by a rapidly growing amount of digital content available on the Web (see also Sect. 2.2.2). There are estimations which claim that more than 90% of the currently produced information is in digital format and that this percentage will likely increase in the future [Var05]. Especially the e-learning area – where content is at the core of the business – shows that on a larger scale content reuse can be difficult to handle, e.g., a developer not only has to find the desired content component, but also adapt it to “fit” into the specific context and other existing content. If a component does not fit, adaptations are necessary that in some cases can be just as tedious as developing everything from scratch. Moreover, as described so far, there are technical and economic reasons to create variants of digital information products, which demand for an approach that is able to handle similar sets of related content within a product line of digital content.

Although there are currently different types of systems which aim to improve content reuse with different methods, e.g., content management systems, document management systems, or workflow management systems (see Sect. 2.2.5), they largely fail to efficiently handle reuse in different information products with content that is in some parts similar and in some parts different, and take advantage of synergy effects. However, even if such systems had the full-fledged mechanisms for product line support, a reuse strategy and a process model would be additionally needed for successful content reuse. The problems arising when a reuse strategy is lacking have been illustrated for content within learning objects in the e-learning area (Chapter 3) where the currently employed reuse techniques are hardly efficient.

In the context of digital information products, current content reuse approaches have got drawbacks in the following respects:

- **Standards:** there are numerous metadata standards for various fields, such as for digital libraries or e-learning, to name just a few (see also Sect. 3.4); this complicates retrieval of content components. Some standards (e.g., in e-learning) seem to be still immature [POS05].
- **Compatibility:** retrieved content components may not be "compatible" with each other or with the ones already owned, i.e., they may have a different file format/encoding, structure, layout, presentation, or assume a different reuse context (see also [GM05]).
- **Strategy:** an overall reuse strategy for content, which goes beyond opportunistic reuse, is frequently missing. The often practiced reuse of content in an opportunistic "copy-and-paste" manner can lead to redundancy and difficult updating.
- **Context:** it is neglected that content may not be developed in isolation.
  – There can be *sets* of digital information products that contain common parts (e.g., some basic subject matters covered in all products, a corporate design for all information products in a product line, etc.) and parts in which they differ.
  – There can be *constraints* on the allowed configurations of variants, which can be derived from an overall strategy (e.g., if a certain subject is covered, then another subject should not be covered).

To solve these problems, a complete "solution package" is needed, which consists of a strategy, a process model, and a tool or system to support the creation of digital information products.

5.3 Requirements on the Approach

Based on the previous expositions, the most important (functional and non-functional) requirements for reuse of content with digital information products are derived in a more general way as follows:

1. The approach should have an *overall strategy for reuse.*
   a) The reuse should be planned *proactively*, before a component is reused.
   b) The approach must have a *uniform view* on reusable content components, and should be able to handle different types of content when applied in different fields.
   c) The approach must be aware of *common parts* and differing parts of content as well as of *variants of digital information products* in a product line.
   d) The approach must be able to handle *constraints upon variants* of digital information products in a product line.
2. A *general process model* is needed which complements the strategy.
3. *Tool support* is needed which helps to enact the well-structured parts of the general process model.

   a) In particular, a tool must permit the assembly of digital information products from existing digital content components.
   b) The tool must be able to model and enact (i.e., execute) the corresponding construction workflow.
4. *Synergy effects* between digital information products in the product line should be exploited. The *development effort* for digital information products with the product line (i.e., with joint domain engineering for all products and individual application engineering for each of the products) should be less than developing all products individually.

The strategy for reuse (1) intends to move away from opportunistic reuse and help to resolve the retrieval and compatibility problems for pieces of content. Furthermore, the focus is explicitly adjusted to sets of digital information products, which accounts for the shortcoming that content may not be developed in isolation. The general process model (2) and the tool support (3) should help to realize the strategy. The exploitation of synergy effects (4) should reduce the development effort. On the one hand, the effort can be assessed in a qualitative way, e.g., if synergy effects indeed reduce redundant parts and allow easier updates in the product line approach compared to an individual development. On the other hand, quantitative effort can be assessed by measuring the development effort, for example, in person months.

5.4 Overview of PLANT

The **Product Lines for Digital Information Products (PLANT)** approach proposes to remedy the aforementioned shortcomings encountered during the creation and reuse of content in digital information products by applying and extending concepts from software product lines. PLANT is a special-purpose approach aimed at the creation of a set of several related digital information products. Each product in a product line is regarded as a variant with a special content configuration, and the creation of information products is standardized within a product line. PLANT is conceived as a "solution package" that encompasses several elements: a strategy, a process model, and appropriate tools. An overview of the elements and their interconnections is presented next.

5.4.1 Strategy and Assumptions

PLANT is designed to be applicable in situations in which several similar digital information products are to be developed. The main strategy is to tackle the aforementioned shortcomings and make the reuse of content more efficient by introducing several limitations known and communicated in advance:

- All developed digital information products must belong to exactly one product line.

- The product line constrains in advance the possible content configurations for different variants of digital information products, defines a fixed set of available content components, as well as the construction process how to create a digital information product out of content components.
- Planning, especially of the content component structure and reuse context, is done (proactively) in advance.

Since the appearance of digital information products may vary in different application areas, PLANT is conceived as a more general approach which can be partly customized; this is done by creating construction artifacts that are customized to fit in the particular area. Furthermore, PLANT is based on the following assumptions:

- Newly developed content often builds upon existing content.
- Newly developed versions of digital information products are often variations of existing ones, having some identical parts.
- The variability of distinct parts is in most cases small or predictable.

Having made explicit the strategy and assumptions, it becomes obvious that PLANT follows a reuse approach that is targeted at a special niche in the content reuse landscape. Contrary to a general-purpose or opportunistic reuse approach, the introduced limitations help to establish a technical and organizational platform for the creation of related digital information products, which can better exploit existing commonalities and achieve synergy effects. The assumptions also make clear when PLANT is *not* applicable. A more detailed process model is introduced next.

5.4.2 A Process Model with Three Sub-Processes

The process model complements the strategy and provides more concrete guidance during the creation of product lines for digital information products. The process model of the PLANT approach is depicted in Fig. 5.1. It is derived from the insights gained with software product lines (Sect. 4.2) and has three different, interrelated sub-processes: family engineering, domain engineering, and application engineering. The sequence of the respective activities in the sub-processes does not have to be strictly waterfall-like, and iterations are possible. An overview of the sub-processes is shown next in Tab. 5.1–5.4. More details will follow in Chapters 6–8, and details on the interrelation of the sub-processes will be explained as well.

Family engineering (1) has the view on the product line as such and provides an organizational roof for the PLANT approach. It is assumed that the activities of family engineering (cf. Tab. 5.1) are executed when a product line is initiated, and thereafter for controlling purposes at specified time intervals or after the completion of one pass of domain or application engineering. The main purpose of family engineering is to complement the purely technical view

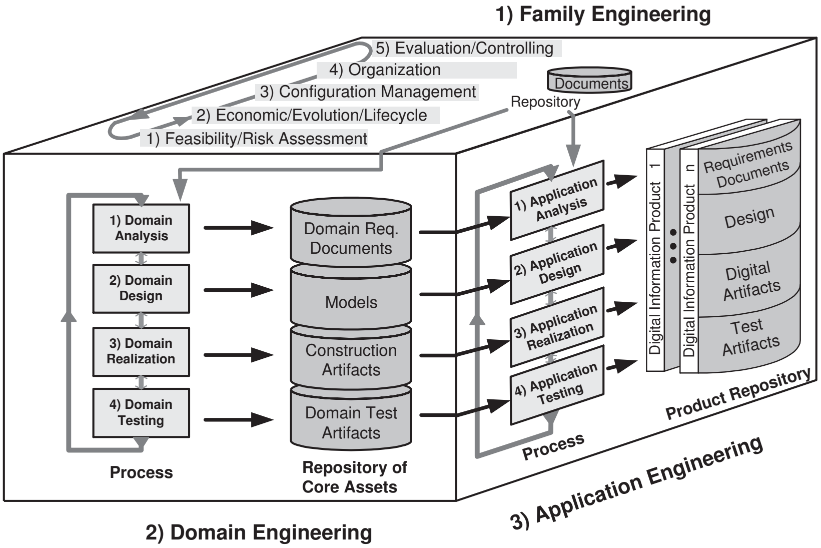


Fig. 5.1. The process model of the PLANT approach with the most important activities and connections.

of domain and application engineering by management issues. More details are presented in Chapter 6.

Domain engineering (2) is a sub-process that focuses on all digital information products in one product line in one specific domain. Most of the overall effort is invested here to plan and prepare reuse. All artifacts created in domain engineering are called *core assets*, and are stored in a *core asset repository* on which activities in other phases can draw upon. The core assets represent the building blocks that can be used later to build a concrete digital information product. There are various types of core assets in PLANT which are shown in Tab. 5.2. The typical inputs, outputs, and activities of domain engineering are depicted in Tab. 5.3.

Domain engineering makes preparations for application engineering and among these, defines exactly one conceptual product line model for one product line. This model specifies *before* reuse which content configurations are allowed for digital information products. Later, in application engineering, one concrete digital information product can only have a content configuration that can be derived from the conceptual product line model; a predefined product map template is used to capture in application engineering the product-specific configurations.

It has already been mentioned that digital information products can consist of a mixture of content and software (Sect. 2.2.5), and this becomes obvious in PLANT since especially the construction artifacts can be content, or a mixture of both content and software; thus, the phases in the domain

Table 5.1. Overview of family engineering in PLANT. A detailed description is given in Chapter 6

1.1 Feasibility/Risk Assessment	
Inputs:	results of interviews/surveys, information about legal regulations, best practices in the field, competitors, current state of market, available resources and constraints
Activities:	assess technical and organizational feasibility of product line; assess risk
Outputs:	family engineering overview document; if necessary: other documents (surveys, etc.)
1.2 Economic/Evolution/Lifecycle Aspects	
Inputs:	overview document
Activities:	estimate effort in person months with/without PLANT; plan update cycles for content during lifecycle of information products
Outputs:	updated overview document; if necessary: other documents with details
1.3 Configuration Management	
Inputs:	overview document; if available: feedback from domain analysis (2.1) with configuration data from market-based view, feedback from application realization (3.3) with configurations from product-based view
Activities:	if data is available: compare and assess if configurations from market-based view and product-based view match sufficiently
Outputs:	updated overview document
1.4 Organization	
Inputs:	overview document
Activities:	choose appropriate organization structure for employees or check whether existing one is still suitable; define who is responsible for family, domain, or application engineering
Outputs:	updated overview document
1.5 Evaluation/Controlling	
Inputs:	overview document
Activities:	check is defined indicators in 1.1–1.4 are within expected ranges; make decisions
Outputs:	updated overview document with next date for evaluation

engineering process are not only concerned with content development, but also in parallel with software development. Content components are construction artifacts that contain a predefined piece of content. Other artifacts such as applets (e.g., used within a Web page) or helper programs, are software. Helper programs are used to create information products by composition out of content components, or to accomplish other tasks; they are procured or created in domain engineering.

A version graph model is defined uniformly for all core assets, but it is employed only if an asset needs to be versioned. For example, a version graph can be used to track the evolution of content components and ease the simultaneous usage of different component versions in different information products.

Table 5.2. Types of core assets for one product line in PLANT. Details are presented in Chapter 7.

1. domain requirements documents
2. models[1]
   a) 1 product line metamodel
   b) 1 conceptual product line model (derived from product line metamodel)
   c) 1 product map template
   d) 1 component model for content components
   e) 1 uniform versioning scheme for core assets
   f) 1 construction specification
   g) $0 \ldots *$ reusable workflow modules
   h) $0 \ldots *$ models for software artifacts (e.g., for helper programs, etc.)
3. construction artifacts
   a) content components
   b) other artifacts (such as layout description files, helper programs, programs that are part of the content)
4. test artifacts

Furthermore, a uniform component model is specified for all content components in one product line, which guarantees that all content components have for example the same structure and the same underlying assumptions for reuse. Therefore, the assembly of content components becomes less problematic than with components found in public repositories which were built according to different requirements.

The construction specification defines in fact a construction workflow model that creates all information products in one product line, using construction artifacts. Reusable workflow modules, which represent prefabricated pieces of a workflow model, can be prepared to speed up the creation of a construction specification. The actual execution of the workflow model is done in application engineering. Additional details on domain engineering of PLANT are presented in Chapter 7.

Application engineering (3) focuses on the construction of one single digital information product (see Tab. 5.4 for an overview of inputs, outputs, and activities). However, it is not allowed to realize every conceivable product, but only one that can be derived from the specifications made in domain engineering. If some specific core assets are needed for a product which are not already there, then application engineering must be stopped and domain engineering is repeated with adequate feedback. The term “application” seems suitable in this context, since (as already argued) the content inside a digital information product may consist of a mixture of content and software.

[1] “1” means “exactly one”; “*” means arbitrary many.

Table 5.3. Overview of domain engineering in PLANT. A detailed description is given in Chapter 7

2.1 Domain Analysis	
Inputs:	results of interviews/surveys; if available: external domain requirements documents (e.g., legal regulations) or other available artifacts; if available: feedback from domain testing (2.4) or application analysis (3.4)
Activities:	define scope of the product line; define content & packaging requirements valid for all information products in the product line; define requirements for additionally employed software artifacts (e.g., helper programs); communicate market-based view of configurations to (1.3)
Outputs:	domain requirements documents
2.2 Domain Design	
Inputs:	domain requirements documents from domain analysis (2.1); if available: feedback from application design (3.2)
Activities:	derive product line models (see Tab. 5.2, 2a-2g) from product line requirements; if necessary: create models for additional software artifacts (e.g., helper programs; see Tab. 5.2, 2h)
Outputs:	models
2.3 Domain Realization	
Inputs:	models from domain design (2.2), if available: feedback from application realization (3.3)
Activities:	realization of construction artifacts for digital information products (Tab. 5.2 3a-3b), i.e., content components (by forward engineering or re-engineering from existing material), realization of other artifacts (e.g., layout descriptions; helper programs for construction, packaging, delivery; programs for test case generation; programs that are part of content), realization of product map database and construction workflow model
Outputs:	construction artifacts
2.4 Domain Testing	
Inputs:	construction artifacts (from 2.3), domain requirements documents (from 2.1), if available: feedback from application testing (3.4)
Activities:	create reusable domain test cases; test construction artifacts (e.g., syntax checking, content validation, test of helper programs), verify that construction workflow model can generate only products with desired configurations, integration testing of content components
Outputs:	tested construction artifacts; reusable domain test cases; if necessary: feedback to domain analysis (2.1)

Due to the preparatory work of domain engineering (e.g., content creation or programming), the focus of application engineering is shifted towards a derivation of a content configuration, and assembly of appropriate core assets. The product map template records the configuration special to one product. The configuration data is stored in a relational database. In a well-designed product line the effort of application engineering should be less than in tradi-

Table 5.4. Overview of application engineering in PLANT. A detailed description is given in Chapter 8

3.1 Application Analysis	
Inputs:	domain requirements documents (from 2.1); if needed: domain design models (from 2.2) for assessing application-specific requirements; results of application-specific interviews/surveys; if available: feedback from application testing (3.4)
Activities:	define particular requirements for one digital information product; consider requirements/constraints defined in domain engineering (2.1 and 2.2)
Outputs:	application requirements documents for one digital information product; if needed: feedback to domain analysis (2.1)
3.2 Application Design	
Inputs:	application requirements documents from application analysis (3.1), models from domain design (2.2)
Activities:	derive the models and the specific configuration for one specific digital information product
Outputs:	application-specific models; if necessary: feedback to domain design (2.2)
3.3 Application Realization	
Inputs:	models from application design (3.2), construction artifacts prepared in domain realization (2.3)
Activities:	configuration and assembly of one digital information product; communication of created configuration (product-based view) to family engineering (1.3)
Outputs:	one digital information product with its specific digital artifacts; if necessary: feedback to domain realization (2.3)
3.4 Application Testing	
Inputs:	the digital information product (from 3.3), domain requirements documents (from 2.1), application requirements documents (from 3.1), domain test artifacts (from 2.4)
Activities:	test and validate digital information product
Outputs:	tested and validated digital information product; if necessary: feedback to application analysis (3.1), domain testing (2.4)

tional single-product development – the payback of the investment in domain engineering.

After a product is created, it is allowed to make some manual adaptations to make it suit better to product-specific requirements, but only if they do not contradict the overall product line constraints, and if the effort of the modifications is low compared to the overall effort needed for application engineering (e.g., adding some cross-references, etc.). However, in a well-designed product line such adaptations should be rare. For major modifications (e.g., of new core assets), PLANT requires a new pass of domain engineering. More details on application engineering are discussed in Chapter 8.

An overview of the models used in PLANT in domain and application engineering is shown in Fig. 5.2, along with a categorization into conceptual, logical and physical design (i.e., from a general to a more specific, implementation-dependent view). The models are on different abstraction layers and allow a delay of design and configuration decisions for a concrete variant up to the latest possible point. At the most general level, the conceptual product line model, which is built according to the product line metamodel, specifies which content configurations are possible for information products. The content component model specifies the general characteristics of reusable content components, which are a special category of core assets of a product line. Core assets can be versioned according to the core asset version graph model. A product map derives a product model based on the conceptual product line model; the product model records a special feature configuration, as well as the versions of the core assets implementing those features. In a general sense, the product model can be regarded as the architecture description of a product. The data of the product map is stored in a relational database, which is queried during the construction of a product. The construction specification is a workflow model that specifies how to create each information product in the product line. Reusable workflow modules are prefabricated model pieces that can be employed to speed up the design of the construction specification model. Additional details on the presented models will be provided in Chapters 7– 8.

How PLANT is initiated

The interconnections between domain, application, and family engineering for digital information products pose the question of where to start when the PLANT approach is initiated. In brief, the initial pass is sketched in Fig. 5.3; PLANT starts in family engineering. If there is a decision in favor of one product line, the process can move on to the first pass of domain engineering. Thereafter, a base of core assets is available for one product line. Individual information products can then be derived in application engineering. After each pass of domain or application engineering, the product line is re-evaluated in family engineering which can use updated data, for example data from domain analysis, or product configuration data recorded in application realization.

5.4.3 Tools and Customization

PLANT is intended to be used in different areas where different types of content exist. PLANT therefore has fixed parts and adaptable parts on different abstraction levels. What always remains fixed are the product line metamodel, the core asset version graph model, and the database schema of the product map template. What can be adapted to a specific area are: the conceptual

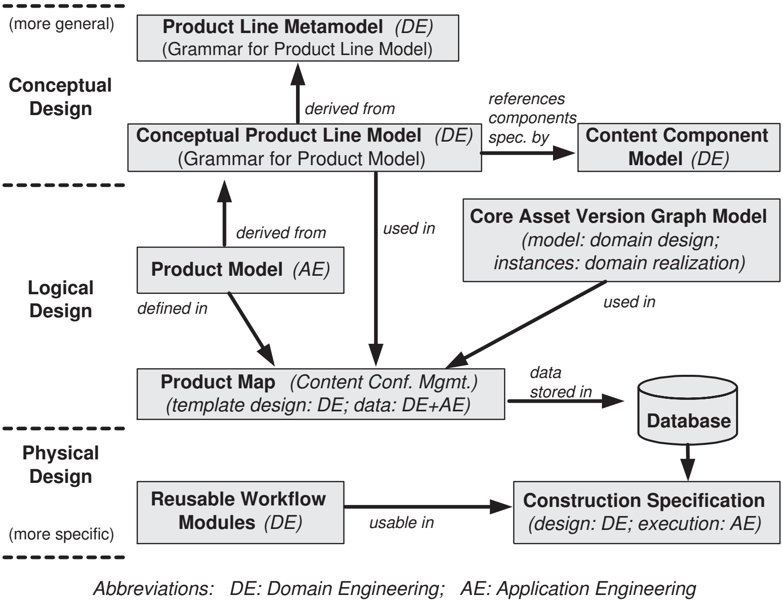

(more general)
Conceptual Design
Logical Design
Physical Design
(more specific)
Product Line Metamodel (DE)
(Grammar for Product Line Model)
derived from
Conceptual Product Line Model (DE)
(Grammar for Product Model)
references components spec. by
Content Component Model (DE)
derived from
used in
Core Asset Version Graph Model
(model: domain design; instances: domain realization)
Product Model (AE)
defined in
used in
Product Map (Content Conf. Mgmt.)
(template design: DE; data: DE+AE)
data stored in
Database
Reusable Workflow Modules (DE)
usable in
Construction Specification
(design: DE; execution: AE)
Abbreviations: DE: Domain Engineering; AE: Application Engineering

Fig. 5.2. Overview of the models used in the PLANT approach.

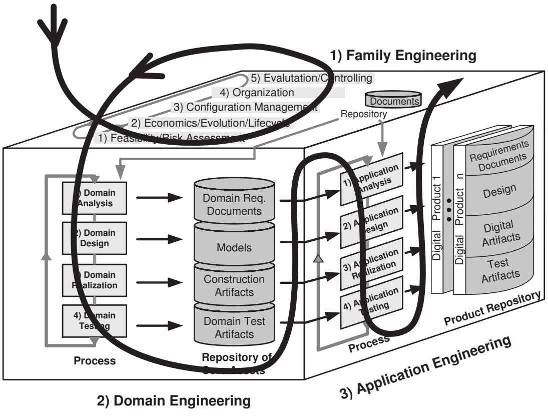


Fig. 5.3. How the PLANT approach is initiated.

product line model, the data of the product map, the construction workflow model, the content component model, the construction artifacts (e.g., content components, helper programs), and test artifacts.

PLANT comes initially with one tool that is able to handle such adaptations, and which mainly targets workflow execution: the Desktop Workflow Engine (DWE). It is used to model and execute the concrete workflow that assembles digital information products from the specific core assets developed in domain engineering. The assembly is done by calling area-specific helper programs that for example construct products by appending Powerpoint files, converting files into different formats such as HTML or PDF, performing XSLT transformations, etc. Additional details on this tool are presented in Chapter 9.

5.4.4 Parallels between PLANT and Software Product Lines

Besides distinction of domain and application engineering in the development process, the product line of digital information products in PLANT has basically the same components that were illustrated for software products lines in Def. 4.4: a set of features, a feature tree, a set of products, a set of core assets, and a construction specification.

However, existing methodologies of software product lines (cf. Chapter 4) cannot be applied to the context of information products right away, because for software product lines the aforementioned components have different assumptions and are tailored to pure software only. As typical examples, features and feature tree focus on the potential functionality of a software system, products are programs, many of the core assets are code components, and the construction specification is used to produce executable programs.

Adaptation is needed for the context of digital information products, especially for capturing the "information" aspect, i.e., which information a product should deliver to a user. Roughly described, features are in PLANT modules of information, the feature tree is extended to a conceptual product line model, the set of products corresponds to the set of digital information products in a product line, the core assets are – in a more general sense – digital artifacts (e.g., content components) that are used for the construction of information products, and the construction specification is a workflow model which specifies how information products are assembled. The parametrization is supported in PLANT by the workflow model, and parameter data is stored in the aforementioned the relational database (cf. Fig. 5.2).

Contrary to the presentation of software product lines, which are intended to be applicable only for software, the PLANT approach is intended to be applicable in several information-product-related areas (e.g., for e-learning, e-news, audio- or video-based information products) and which will not be formalized each, as such a general formalization would be necessarily incomplete. In the sequel, the general PLANT models are specified in a formal way, to help express their exact semantics and assess the expressive power. Further connections between PLANT and the definition of a software product line (Def. 4.4) as well as extensions will be discussed in the appropriate parts.

5.4.5 Examples for Application Areas

The PLANT approach is meant to be applicable in different areas in which digital information products play a role. Four important areas are sketched in the sequel, along with the respective motivations, the meaning of the conceptual product line model, the area-specific construction artifacts (i.e., content components or helper programs), the purpose of the construction specification, and resulting products. The examples should clarify how and to which degree PLANT can be customized to a specific area.

Product Lines for E-Learning Courses

- **Motivation:** create variants of an e-learning course (e.g., “Information Systems”) with a different emphasis on sub-topics from the database or workflow area, depending on the target audience.
- **Conceptual product line model:** models commonalities, differences, and valid content configurations for all relevant course variants.
- **Core assets:** content components: Powerpoint files for subtopics (e.g. “ER model”, etc.) in PPT format; layout descriptions: PPT-Master files; helper programs: a program to append two given PPT files to one file; a program to apply given a given PPT layout to a given PPT file
- **Construction specification:** models how to append which files to obtain a certain course; the decisions which path to take during execution lead to one course with a specific configuration for one target audience
- **One generated product:** a PPT handout that is sent to students.

Product Lines for E-Books

- **Motivation:** efficient creation of PDF files for the user manual of similar digital camera models. As such cameras themselves have technical commonalities and differences, their corresponding documentation will necessarily have common parts and differing parts.
- **Conceptual product line model:** models commonalities, differences, and valid content configurations for user manuals.
- **Core assets:** content components: chapter texts in XML format; layout descriptions: XSLT and XSLT-FO specifications [HM04]; helper programs: an XSLT parser that generates PDF files from the content components according to the XSLT specifications.
- **Construction specification:** models when to call an XSLT parser with which parameters; specifies how the resulting artifacts are copied on a Web server.
- **One generated product:** a PDF file representing the documentation for a specific camera model.

Product Lines for E-News

- **Motivation:** package and deliver the news according to the interests of different target groups (e.g., general interest, business, science, etc.). To maximize the profits, only certain configurations should be allowed.

- **Conceptual product line model:** models the relevant domains of news and selectable content (e.g., foreign affairs); the allowed configurations are based on the business model.
- **Core assets:** content components: HTML files with a standardized structure; cascading style sheets (CSS) define layout specifications; helper programs: program to generate a Web site with a corresponding menu structure. The core asset version graphs of the content components are used to capture the versions of content (associated to a feature like "foreign affairs") from different days.
- **Construction specification:** call the programs that create the Web site and publish the corresponding files on a Web server.
- **One generated product:** a Web site that contains a special configuration of news content.

Product Lines for Audio-based Products

- **Motivation:** instructional podcasts [Coc05] are available on the Web, such as spoken encyclopedia articles. For a certain topic (e.g., history of computing), sensible variants can be chosen that suit the needs of a listener or a group of listeners.
- **Conceptual product line model:** meaningful configurations of audio chapters
- **Core assets:** content components: audio files in MP3 format that cover a certain chapter; a helper program that can insert a jingle (short melody) at the beginning of each chapter to audibly mark the beginning of a new topic, a helper program to append MP3 files one after another.
- **Construction specification:** construct a specific audio file (based on the selection constraints of the feature model) by appending existing files one after another and inserting jingles at defined points.
- **One generated product:** a podcast covering a meaningful combination of selected topics.

5.5 Validation Approach

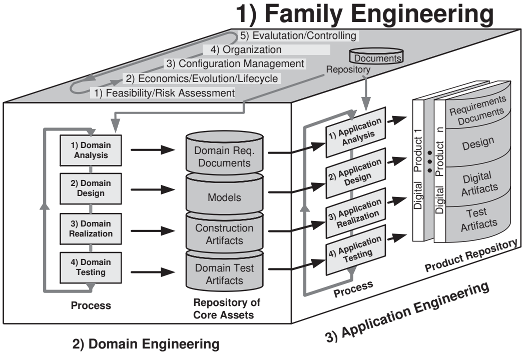
One important question is how it can be assessed whether the presented approach indeed solves the aforementioned problems in practice. To show the plausibility of the proposed solution, various *scenarios* are used throughout this thesis from the e-learning field. As already pointed out in Chapter 3, one can observe that information in e-learning field is currently represented by a multitude of different content types, such as text, graphics, animations, audio and video files, or programs (e.g., Java applets). It is assumed that if the presented approach is usable in such a complex environment, it can also be used in other environments with similar or less complex content. Therefore, e-learning will be used in the sequel as a typical field for the application of digital information products. After the introduction of the details of the PLANT approach, a *case study* will additionally show in Chapter 10 how PLANT was practically applied in a real-world context related to e-learning.

5.6 Summary and Discussion

This Chapter presented a macroscopic overview of the **Product Lines for Digital Information Products (PLANT)** approach which targets a special niche in the content reuse landscape for information products. In PLANT, a digital information product is understood to be a variant with one of several predefined content configurations. The product line as such is, from a more general point of view, a kind of envelope that imposes predefined constraints upon all its digital information products. These constraints are known and communicated in advance, and are expressed in the aforementioned documents and models. PLANT attempts to make content reuse feasible and more efficient by achieving synergy effects. These can reduce development times and yield higher-quality information products due to less redundancy and simplified updates. More details on the three sub-processes of PLANT – family engineering, domain engineering, and application engineering – will be presented in the following Chapters.

6

Family Engineering in PLANT



The family engineering sub-process in PLANT has the view on one product line as such, and provides an overall organizational roof that is intended to complement the technical views of domain and application engineering by management issues. Family engineering is carried out once during the initial creation of a product line, and indicators with specified ranges are predetermined for various aspects. In addition, family engineering is executed for controlling purposes at predefined time intervals or after the execution of domain or application engineering. Within family engineering, the following aspects play a role: feasibility and risk of creating a product line, economic aspects, a comparison of market-demanded and technically realized product configurations, organizational aspects, and controlling aspects. This Chapter discusses each of these aspects in a detailed way, and finally points out some of the typical difficulties.

6.1 Feasibility and Risk Assessment

An important aspect at the initial creation of a product line is whether it is feasible to create it, and the involved risks. These aspects can be assessed in various ways, qualitatively (e.g., through questionnaires) or quantitatively (e.g., through measurements or estimations). Possible inputs for such an assessment can be: results or interviews or surveys, legal regulations or other constraints, descriptions of best practices, data about competitors or markets, or information about the availability of needed resources.

To keep a better overview and simplify controlling afterwards, PLANT distills the most important information from the aforementioned documents into a *family engineering overview document* which is created when a product line is initiated, and which contains general information about the product line (e.g., identifier, description, domain of intended use), as well as information from other activities in family engineering. In the family engineering overview document, the data characterizing feasibility and risk of a product line is summarized with respect to predefined factors which are considered most relevant (derived from the assumptions in Sect. 5.4.1), using a rating approach. For each factor, there is an ordinal scale with 5 possible answers a)–e), from which exactly one must be chosen[1]. The scale is designed in such a way that if all answers are either d) or e), then feasibility and risk of a product line are within acceptable ranges. If any of the answers is either a) or b), then PLANT should not be used. If any of the answers is c), then the development should not go on until the respective issue is clarified to be either in a), b) or in d), e).

Feasibility

- **1) Is the product line technically feasible?**
  a) infeasible; *b*) almost completely infeasible; *c*) unsure; *d*) feasible, but with restrictions; *e*) completely feasible
  Aspects to be considered: e.g., can the relevant file formats be handled? Are the necessary tools, programs, etc., there? Can the missing tools or programs be technically realized? Is the technical infrastructure available?

- **2) Do information products in the product line have commonalities?**
  a) no commonalities; *b*) some commonalities, but which cannot be technically exploited; *c*) unsure; *d*) some commonalities that can be technically exploited; *e*) many commonalities that can be technically exploited
  Aspects to be considered: e.g., is there enough potential for synergy effects? Are there content similarities that can be modularized?

[1] This scale is similar to a Likert-like scale [Lik32] that is used to measure strengths of agreement.

- **3) Are the variable parts of information products known?**
  a) unknown or unpredictable; *b*) only partly known or unpredictable; *c*) unsure; *d*) most known, rest predictable; *e*) all known in advance
  Aspects to be considered: e.g., for the parts that are individual to each product and that are not in common with other products, is it possible to find some regularities?

- **4) Is the product line organizationally feasible?**
  a) infeasible; *b*) many arguments against it; resources not available; *c*) unsure; *d*) feasible, some management support; *e*) feasible, strong management support
  Aspects to be considered: e.g., are the necessary resources available (staff, etc.)? is the strategy of product lines clearly communicated and does the management support this?

The questions on risk assessment focus on the factors considered most important in the PLANT context, and for this context, risk is understood as the degree of uncertainty which is influenced by several factors. In addition, explanations can be given in natural language to show possible consequences associated with a certain factor.

Risk assessment

- **1) In the domain where PLANT is to be applied, how often are radical changes expected to occur?**
  a) often; *b*) very likely; *c*) unsure; *d*) rarely; *e*) never
  Aspects to be considered: e.g., radical changes of the domain (in which information products will be created) can affect what is common and what is variable in different products. If such changes occur too often, the preparations done in the product line could not pay off.

- **2) Is the demand for information products predictable?**
  a) not predictable at all; *b*) hardly predictable; *c*) unsure; *d*) predictable; *e*) known for sure
  Aspects to be considered: e.g., if the demand (internal in a company or external from customers or a market) is unstable, then the investment in a product line is more risky than if the demand is stable, because the preparations to create products may become obsolete

- **3) Are there strategic advantages that are expected from a product line approach?**
  a) no advantages expected; *b*) overall strategy is unclear; *c*) unsure; *d*) strategy is in most parts defined and advantages are expected; *e*) strategy is defined and advantages are expected
  Aspects to be considered: e.g., a product line should be created only with

a clear goal and strategy where it is to be applied and why (efficiency aspects, competitive advantage, entry barriers for competitors, etc.). Otherwise, technical and organizational aspects cannot be aligned towards a certain goal, which increases the risk of failure.

In addition to the factors mentioned before, brief textual comments can be supplemented in an additional section in the family engineering overview document. These can characterize other aspects related to feasibility or risk, for example, if some digital artifacts are already existing, names of competitors, etc.

6.2 Economic, Evolution, Lifecycle Aspects

To assess whether the application of PLANT has economic advantages, one has to quantitatively compare the situation with PLANT and without PLANT. In particular, PLANT uses as a basis for this comparison the effort in person months, and it is assumed that this quantity can flow as well into the computation of other economically relevant quantities (e.g., costs, net present value, etc.).

At the initiation of a product line, most values for the situation with PLANT inevitably have to be estimated, and only those for the situation without PLANT may be already known (e.g., by measuring the status quo values). However, when family engineering is repeated at predefined time intervals or after application engineering, the estimations can become more accurate and thus improve their validity.

Closely related to the economic aspects are also the evolution and lifecycle aspects of products, as it can be expected that the content in digital information products will evolve over time and will require updates. The strategically planned lifecycle of information products influences the frequency of needed updates. The evolution and lifecycle data that is most important from an economic point of view is collected here; other aspects of evolution and lifecycle (e.g., technical configurations) are managed in an integrated way during domain and application engineering (see Sect. 7.2.2).

6.2.1 Estimating Effort with PLANT

The effort estimation with PLANT is derived from a model that has already been developed for the context of software product lines ([BCM$^+$04]; see Sect. 4.5.1). The effort with PLANT (in person months) is calculated as follows:

$$E_{with} := E_{org} + E_{cab} + N * (E_{reuse_{with}} + E_{unique_{with}} + j * E_{update_{with}})$$

with

- E_{org}: *effort to introduce the product line, adapt the organization, train staff, etc.;*
- E_{cab}: *effort for the development of a core asset base, cost of the associated commonality and variability analysis;*
- N: *number of digital information products in the product line;*
- $E_{reuse_{with}}$: *average effort (for one information product) for reuse of existing core assets, e.g., choosing configuration, searching, checking, integration;*
- $E_{unique_{with}}$: *average effort to extend the core asset base with core assets unique to a product, effort for manual adaptations after generation;*
- j: *average planned number of content update cycles for one information product;*
- $E_{update_{with}}$: *average effort of updating the product-related core assets in the core asset base;*

In a well-designed product line, $E_{reuse_{with}} + E_{unique_{with}}$ should be relatively small[2], compared to $E_{org} + E_{cab}$.

6.2.2 Estimating Effort without PLANT

In the case without product lines, the effort to create N individual products is:

$$E_{without} := N * (E_{unique_{without}} + j * E_{update_{without}})$$

with

- N: *number of individual digital information products;*
- $E_{unique_{without}}$: *average effort to create one unique information product;*
- j: *average planned number of content update cycles for one information product;*
- $E_{update_{without}}$: *average effort of updating one information product;*

6.2.3 Comparison of effort with/without PLANT

The economic rationale of a product line of information products is similar to that of software product lines (see Sect. 4.5.1). If the product line for information products is well-designed, then $E_{reuse_{with}} + E_{unique_{with}} < E_{unique_{without}}$, and $E_{update_{with}} < E_{update_{without}}$. However, with PLANT, one must first invest in $E_{org} + E_{cab}$, which will typically pay off only after several information products are realized. If synergy effects can indeed be realized with PLANT, then $E_{with} < E_{without}$ must be satisfied. Only then should the PLANT approach be applied or continued to be applied.

[2] What “small” means depends heavily on the specific domain and thus cannot be specified precisely in a general way.

6.3 Configuration Management

Configuration management for digital information products is similar to software configuration management, and also has the concepts of configuration items, configurations, and baselines [Tic92]. In general, *items* are artifacts that are the smallest unit of change. A *configuration* is a set of items, and the set is fixed at a certain point in time. A *baseline* is an unambiguous description of a configuration, identifying which items belong to a particular configuration.

In PLANT, configuration management has two views: the *market-based view* and the *product-based view*. The main reason for this splitting is that not every configuration that is demanded by a market (or customer) will or can be technically realized. From a management point of view, it is therefore important to keep an eye on potential mismatches to avoid that the product line technically evolves to a point where none of its products is demanded any more, and work against such effects. In fact, the data for market-based view of configuration management results from domain engineering, and that of the product-based view from application engineering. Configuration management in family engineering is merely an integrated view to evaluate mismatches. When the product line is initiated, there is no configuration data there yet, and this step is overridden; this step becomes important during controlling.

6.3.1 Market-Based View

The market-based view of configuration management captures from the demand side, i.e., from a market or customer perspective, which potential content components are demanded for different information products. Therefore, the configuration items are the demanded content components, a configuration is a fixed set of demanded components for one information product, and a baseline is a description defining which demanded components make up a certain information product. The data for the market-based view of configuration management is obtained from domain analysis (see Sect. 7.1).

6.3.2 Product-Based View

Broadly speaking, the product-based view of configuration management captures which versions of which artifacts are used to implement which information product. In the technical context of information products, the items to be configured are core assets, like for example content components. Information products are made up of versions of different core assets; such a set of versions of core assets represents a configuration. A baseline is a description of a configuration that precisely defines which core assets make up a certain digital information product. The baseline data is reported by application engineers after the creation of a product.

6.3.3 Matching Market-Based and Product-Based View

The detailed comparison of the market-based view and product-based view of configuration management can become time-consuming even if it is to be automated, as appropriate links between demanded and realized features would have to be created and updated continuously. However, PLANT considers that managers in family engineering do not need too detailed information at this point, but merely a status reported by experts whether the technical configurations match the market-demanded ones to a certain degree, which helps managers to take appropriate actions.

The match between the market-based and product-based configuration is therefore summarized in the family engineering overview document, in a scale similar to those used for feasibility and risk assessment (Sect. 6.1).

- **Match between market-based and product-based view of configuration management**
  a) none of the configurations demanded in market-based view match those in product-based view; *b*) few of the configurations demanded in market-based view match those in product-based view; *c*) unsure – must be clarified; *d*) the majority of the configurations in market-based view matches those in product-based view; *e*) all configurations in market-based view match those in product-based view

The meaning of “few”/“majority”/“match” may be defined individually for each product line in a more precise way. The ideal situation is *e*). In case of *d*), managers should take actions to achieve *e*) if possible, or at least maintain *d*). If the match is rated *a*) or *b*), then managers must take actions to avoid the technical base of the product line to evolve into a wrong direction, which could lead to a situation where standardization comes to a degree in which demand is not satisfied at all.

6.4 Organizational Aspects

PLANT requires in family engineering the definition of key organizational aspects for each product line and summarizes them as well in the family engineering overview document. In particular, a brief statement of the chosen organizational model must be included. For example, the model can be developed individually or based on one of the models presented in Sect. 4.5.2, such as, the development department model, the business units model, the domain engineering unit model, or the hierarchical domain engineering units model. After the choice of the model, the management has to take care of its realization and provide the necessary infrastructure, such as an IT infrastructure.

In addition, the number of staff and the names of the employees that are available for the product line are recorded in the family engineering overview document. Role responsibilities are defined in a general way by assigning every staff member to the categories family engineering, domain engineering, or application engineering; it is possible to assign one employee to more than one category. For some product lines it is possible that there is only one employee who is responsible for all categories, if the effort required for the respective activities is small enough. For more complex scenarios, the role model can be refined to fit the specific situation, i.e., additional, more specifc roles can be defined for family engineering, domain engineering, or application engineering.

6.5 Evaluation and Controlling

The family engineering document provides in the evaluation and controlling section a summary of the issues monitored in the previous phases. It helps managers to make decisions, for example whether to continue with the product line approach or not.

Here, feasibility and risk assessment are each summarized to "ok/critical/to be checked". For example, if all answers related to feasibility are in *d*) or *e*), then feasibility is "ok"; if one of the answers is in *c*), then feasibility is "to be checked"; if one of the answers is in "a)" or "b)", then feasibility is "critical". The same is done for risk assessment. From the economics part, the estimated and actual savings with PLANT are listed. The comparison between market-based and product-based configuration management is summarized (similar to feasibility) to "ok/critical/to be checked".

In addition to this summary overview, additional comments can be added which point to issues that are to be monitored or evaluated regularly, and which may depend on the specific product line. These can be, for example, inspections of whether synergy effects were indeed realized, customer satisfaction obtained from surveys, potential improvements, etc.

It is important that the controlling of the development of the data in the family engineering overview document is done in a repetitive manner in order to be able to take corrective actions. Therefore, this document contains the date of the current inspection and the date of the next planned inspection, where the data of each phase is updated sequentially, and the approach is re-evaluated. The data update or completion, and the re-evaluation are also done after one pass of domain engineering or application engineering.

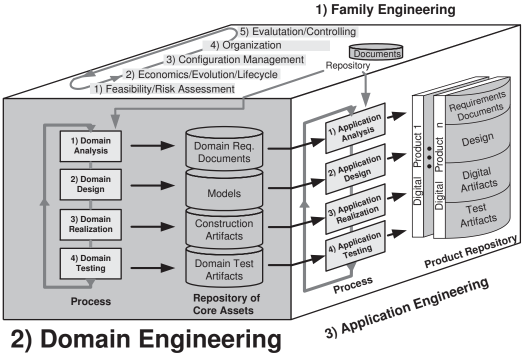
6.6 Summary and Discussion

Family engineering is mainly concerned with management and organizational issues at the initiation of PLANT, and also throughout development. Carrying out family engineering in a repeated fashion at specified intervalls makes it possible to base management decisions on recent data. The family engineering overview document summarizes the most important information related to feasibility and risk assessment, economic aspects, an integrated evaluation of the market-based and product-based view of configuration management, organization, and evaluation and controlling.

One main difficulty of family engineering is that it also has to deal with “soft” factors such as feasibility or risk, which are difficult to describe or quantify, or which could be quantified in different ways. However, such difficulties are independent of this approach as they are inherent to the factors themselves. Although a more complex approach could have been taken for their assessment, the employed ordinal scales in PLANT try to find a balance between required effort and usefulness of the results, which still allows to steer the product line development from the management side.

7

Domain Engineering in PLANT



This Chapter presents the details of the domain engineering sub-process, which is the sub-process that requires the most effort. It has the view on all digital information products of one product line and its purpose is to make all preparations necessary for application engineering.

Starting with domain analysis, the domain that is relevant for the product line is scoped and domain requirements are collected. The results are used in domain design to create more abstract models that address all products in one product line. These models belong to different layers of abstraction, ranging from conceptual design over logical design to physical design. Content components and other implementation-related core assets are implemented in domain realization, and they will serve later as building blocks for the information products to be created in application engineering. Domain testing focuses on the peculiarities of the artifacts prepared for the creation of information products, such as content component testing, verification of the construction workflow model, and integration testing of content components. The Chapter finishes with a summary.

7.1 Domain Analysis

The domain analysis process systematically captures the requirements for all digital information products that will be eventually built. In addition, scoping is performed in a way that is similar to software product lines (cf. Sect. 4.3.1). Domain analysis uses the following *inputs* (cf. Tab. 5.3): results of interviews or surveys, other available domain requirements documents (e.g., obtained from law regulations) or existing artifacts. Furthermore, if domain analysis is not done for the first time and if some digital information products have already been created in application engineering, possible feedback from domain testing or application analysis can be used to derive new or adapt existing requirements.

The output of domain analysis is one or more documents that contain natural language descriptions for scoping, and requirements for content used for information products. Details are presented next.

7.1.1 Domain Scoping

In PLANT, a *domain* is understood to cover a certain area of information, or in a general sense, a field of knowledge. Similar to software product lines (Sect. 4.3.1), scoping for digital information products is split up into three parts that introduce limitations:

- **Product portfolio scoping:**
  Purpose: limits the number of different information products to what is technically feasible and considered relevant for one product line.
  Output: a general statement in natural language of what an information product is, and which variants are generally needed.

- **Information domain scoping:**
  Purpose: defines a hierarchy of information domains that are relevant for all information products in one product line. It is assumed that in one product line there is exactly one root domain that can hierarchically contain zero or more subdomains which themselves might contain zero or more subdomains, and so on.
  Output: description of the hierarchy of information domains.

- **Asset scoping:**
  Purpose: specifies limitations for the core assets used in one product line. For example, the structure, layout, or file format of content can be precisely limited.
  Output: limitations for core assets expressed in natural language.

7.1.2 Capturing Requirements for Information Products

Based on the aforementioned inputs, the content requirements and packaging requirements are captured that are relevant for all digital products in the product line. Additionally, the requirements for the employed software artifacts, such as helper programs, are captured as well.

Content Requirements

Content Requirements are defined and categorized into common, optional, and alternative requirements. *Common requirements* refer to content that has to be included into every digital information product in the product line. *Optional requirements* refer to content that may be realized in some information products, but not in all products. *Alternative requirements* define a set of alternative topics along with a minimum and maximum number that indicate how many topics have to be covered at least/at most.

These categories of requirements implicitly define the demanded content configurations from a demand-side perspective, and thus make up the market-based view of configuration management (see also Sect. 6.3.1).

In addition to the common, optional and alternative requirements, *constraints* can be formulated for different product variants (e.g., in natural language). For example, a constraint can limit the valid configurations by stating that certain content must be covered when another particular content is covered. Furthermore, *content component model requirements* are captured which are relevant for the choice of the uniform content component model. In addition, *crosscutting concerns* are defined, which are requirements that simultaneously affect more than one digital information product. An example of a crosscutting concern is a layout specification which has to be used in every information product.

Packaging Requirements

The packaging requirements describe how the resulting artifacts should be put together for delivery. Conceptually, a package contains one or more artifacts of an information product. Every digital information product is delivered in at least one package. Technically, a package can be for example a folder, a ZIP-file, a learning object, or a file with an application-specific format (e.g., Powerpoint). For a package, a general order can be specified in natural language, of how artifacts representing certain content have to be included.

Requirements for Software Artifacts

Requirements for software artifacts can be captured as in traditional requirements engineering. The methods are well-known and the reader is referenced to [Poh97, Som04b, Pre05] for a discussion. Software artifacts used in the context of information products are for example helper programs that will be used to assemble information products out of content components, or Java applets that are part the content.

Example 7.1

Let’s consider as a brief example for scoping and requirements capturing for a product line of e-learning courses in the domain “Information Systems”:

Domain scoping

- **Product portfolio scoping:** a digital information product is a variant of a course in the area of Information Systems. In this product line, a course covers databases and workflow aspects, or only database aspects.
- **Information domain scoping:** the root domain is “Information Systems” which has two subdomains considered relevant here: “databases” and ”workflow”. The domain “databases” is defined to have the subdomains “database modeling”, “database theory”, and “database implementation”. The domain “workflow” is defined to have the subdomains “workflow modeling” and “workflow implementation”.
- **Asset scoping:** every content component must be a Powerpoint file with the same predefined master layout.

Content requirements

- **Common requirements:** every course must have a general introduction; every course must contain the domains “databases” and “database modeling”; the database domain must have an introduction; “database modeling” must contain the “ER model” and the “relational model”. The domain “workflow” must have an introduction.
- **Optional requirements:** the domain “databases” may contain optionally the domain “database theory” which discusses “normalization”; the domain “databases” may contain optionally the domain “database implementation” which discusses “SQL”. A course may optionally contain the domain “workflow”, which in turn may contain optionally the domain “workflow implementation” discussing “systems architecture”.
- **Alternative requirements:** The domain “workflow modeling” must contain either “Petri nets” or “Event-driven Process Chains”, or both.
- **Constraints:** If the “workflow” area is covered, then the domain “workflow modeling” must be covered.
- **Content component model requirements:** A content component must be realized as a Powerpoint file.
- **Crosscutting concerns:** a uniform layout is to be used for all courses; it is defined in a separate Powerpoint master file.

Packaging requirements: A package is a Powerpoint file (i.e., the package has an application-specific format) which can contain the slides of one or more content components. In total, there must be 3 packages: for an introductory part, for databases, and for workflow. Inside the database package, content components should be included in the following order (or sequence): 1) “introduction to DB”;

2) “ER model” 3) “relational model” 4) “normalization” 5) “SQL”. Inside the workflow package there is the order: 1) “introduction to WF”; 2) “Petri nets”; 3) “Event-driven Process Chains”; 4) “systems architecture”. The order in the packages must be preserved even if some content component is not included.

Requirements for software artifacts

- **Helper programs:** The helper program “pptappend.exe” is called with two Powerpoint files as parameters, p_1 and p_2, and it appends all slides of p_2 after the last slide of p_1. The program “pptapplytemplate.exe” is called with parameters p_1, p_2, where p_1 is a Powerpoint master file and p_2 is a directory that contains Powerpoint files; the program applies the layout defined in the master file to all Powerpoint files in the directory.

□

7.2 Domain Design

The domain requirements documents from domain analysis are used in domain design to create models for the product line that provide more abstraction and ease the development of information products. For the product line, there are different areas for which models are created (see also Fig. 5.2):

- **Conceptual Design**
  - *Conceptual product line model*: incorporates the domains and features available for all information products in the product line. For one specific product, valid configurations can be chosen only from this model.
  - *Content component model*: defines the details for content components, such as file format, metadata, etc.
- **Logical Design**
  - *Core asset version graph model*: handles the evolution of core assets by specifying uniformly how assets are versioned. If a core asset needs to be versioned, this is done according to this model.
  - *Product map template*: structures the configuration management data for information products. It is filled in application engineering with the data which versions of which core assets are used for a particular product.
- **Physical Design**
  - *Construction specification*: defines for the whole product line one construction workflow model that is used to create every possible information product in the product line.
  - *Reusable workflow modules*: are prefabricated and tested pieces of a workflow that can be reused in domain engineering.

The models created in PLANT for the product line have a conceptual design layer that is mostly implementation-independent, while the physical design layer is mostly implementation-dependent. The models in the conceptual design layer do not contain all the details in the beginning (e.g., the sequence in which content components should be assembled). Details are subsequently added in the logical design layer (e.g., versioning) and in the physical design layer (e.g., sequence of content components within a package, and assembly of components).

In addition to the design models for the product line, other design models can be created for software artifacts, such as the helper programs that are to be implemented additionally. These models can use traditional notations like the Unified Modeling Language (UML) [Obj07]; they are well-known and thus will not be discussed here in detail.

7.2.1 Conceptual Design: Defining Allowed Configurations

The conceptual design of the product line has the highest level of abstraction and defines in a general way the valid configurations for digital information products in a product line, as well as a uniform model for content components.

The Conceptual Product Line Model

On a conceptual level, it has to be modeled first which digital information products can be built within the product line and which not, which commonalities and differences are there between possible products, and which configurations are allowed. These aspects are defined in PLANT using a tree-like conceptual product line model.

Constructs

The conceptual product line model consists of non-empty domains which can contain features or other (sub)domains. Conceptually, a *feature* is understood to be an indivisible module of information; from an implementation-oriented view, a feature will be realized by a content component that can consist of a set of digital artifacts. The term “feature” is adopted from the software product lines area, and the content such a feature refers to can consist of more than simple text, but also contain multimedia components or programs such as Java applets (see Sect. 2.2.5). In the conceptual product line model, a domain is merely a construct to group related features or other sub-domains. The sub-domains and their hierarchical relationships are taken from domain scoping (Sect. 7.1.1). Since features are meant to be realized by self-contained content components, PLANT assumes that in domain engineering there are no interactions between domains or between features[1]. Domains or features can be of type *common, optional, alternative.* In the following, features of type common, optional, alternative will be referred to as *atomic features*.

Additionally, a *crosscutting feature* is a feature (with no type) that is merged during the execution of the construction process in application engineering with all atomic features in the domain where it is located, as well as with all atomic features in all of its subdomains. One typical property of a crosscutting feature (e.g., a layout specification) is that after it is merged with an atomic feature, it is not distinguishable within the result as a modular entity. In principle, crosscutting features are comparable to aspects in Aspect-Oriented Programming (AOP) [FECA04, PSV05]; they are also similar to graybox components [AN03][2].

Grammar

Exactly one conceptual product line model has to be defined for one product line. The grammar and the structure of the conceptual product line model, however, are always the same and do not depend on a particular product line. The grammar $G = (N, T, P, S)$ of the conceptual product line model is described more precisely in Fig. 7.1 using the Extended Backus-Naur Form (EBNF)[3]. It can be understood as a metamodel of the conceptual product line model.

In principle, the terminals of the grammar are the actual domains and features. The root domain terminal is denoted as $'RD'$, a *common domain* as $'CD'$, an *optional domain* as $'OD'$, and an *alternative domain* as $'AD'$. The terminals for *common features* are denoted as $'cf'$ (i.e., features of type common), for *optional features* as $'of'$ (i.e., features of type optional), for *alternative features* as $'af'$ (i.e., features of type alternative), and for crosscutting features as $'ccf'$ (i.e., features of type crosscutting). The grammar has similarities with attribute grammars introduced by [Knu68], as it is assumed here that terminals have attributes associated with them; in this grammar,

[1] As already mentioned, it is possible to make minor adaptations in application realization after a product was created. It is therefore possible to add interconnections manually in a specific product, such as cross-references in a text that was generated from content components.

[2] The formalization of a crosscutting feature is omitted, since such a feature and the way how it is merged with atomic features can be different for different application areas and component models. For example, a layout defined in a cascading stylesheet is code that has to be inserted at a specified location inside an HTML page, a layout definition in Powerpoint is a special binary file that can be internally merged by the program, etc.

[3] In brief, the ISO notation [ISO96] for the Extended Backus-Naur Form (EBNF) uses '=' for 'is defined to be'; ';' for end of production; quotes for terminal symbols, e.g., 't' for terminal symbol t; '|' for 'alternatively'; $\{X\}$ for a repetition of zero or more X; () for grouping; '*' for repetition, e.g. 2*'X' means 'XX'. Details see [ISO96].

however, the only purpose of these attributes is to capture the names of domains or features, and the min and max ranges of alternative domains.

- *Non-terminals N* = { *RootDomain, Domain, CommonDomain, OptionalDomain, AlternativeDomain, OtherFeatures* }
- *Terminals T* = { *'RD', 'CD', 'OD', 'AD', 'cf', 'of', 'af', 'ccf', '(', ')'* }
- *Start symbol S* = *RootDomain*
- *Productions P in EBNF notation [ISO96]:*

```
RootDomain        = 'RD' '(' 'cf' OtherFeatures {Domain} ')';           (1)
Domain            = CommonDomain                                        (2.1)
                    | OptionalDomain                                    (2.2)
                    | AlternativeDomain;                                (2.3)
CommonDomain      = 'CD' '(' 'cf' OtherFeatures ')'                     (3.1)
                    | 'CD' '(' OtherFeatures CommonDomain
                      {Domain} ')';                                     (3.2)
OptionalDomain    = 'OD' '(' 'cf' OtherFeatures ')'                     (4.1)
                    | 'OD' '(' 'of' OtherFeatures ')'                   (4.2)
                    | 'OD' '(' OtherFeatures Domain {Domain} ')';       (4.3)
AlternativeDomain = 'AD' '(' 2 * 'af' {'af'} ')'                        (5.1)
                    | 'AD' '(' 2 * AlternativeDomain
                      {AlternativeDomain ')' };                         (5.2)
OtherFeatures     = {('cf' | 'of' | 'ccf')};                            (6)
```

Additional assumptions

- *The terminals $RD, CD, OD, AD, cf, of, af, ccf$ have the attribute* name *which holds a name associated to the respective domain or feature.*
- *The terminal AD additionally has the attributes $min, max \in \mathbb{N}$ with $min \leq max$, and $max \leq$ the number of contained alternative features or alternative domains.*
- *In $L(G)$, a terminal with a particular value of its name attribute must not occur more than once.*

Fig. 7.1. Product line metamodel as the grammar $G = (N, T, P, S)$ for conceptual product line models.

Using the grammar, one can construct a tree that contains features in the leaves, and domains in the inner nodes. The key idea is that the structure of the tree, defined by the grammar, and the types of domains and features impose constraints on which of them can be chosen for a particular digital information product later in application engineering. The semantics of common, optional, and alternative features are similar to those defined in Def. 4.4. Basically, the root domain is incorporated by default into every digital information product. Common features or domains have to be incorporated if their parent node is incorporated. Optional features and domains can be included in some products, but only if their parent node was already included. If the parent node of an alternative domain is included, then a certain number of features of the contained alternative features or alternative domains have to be selected, and this number must be greater than or equal to the *min* attribute, and less than or equal to the *max* attribute associated to the respective domain. Details on how features are selected for a particular information product are explained later in the context application engineering (see Sect. 8).

The sequence of how domains or features can appear in the conceptual product line model is precisely defined by the grammar. However, the details of the exact sequence specifying which feature is implemented in which package after which other feature is postponed until physical design (see 7.2.3). There, the focus is on the exact creation procedure, and the constraints for the creation procedure are derived there from the conceptual product line model.

Graphical Notation

The graphical notation in Figure 7.2 will be used throughout this thesis to depict conceptual product line models for digital information products, which can be drawn as trees. The structure of the model results from the application of the grammar in Fig. 7.1. The notation is a modified UML notation that presents domains as rounded rectangles, and atomic features (i.e., common, optional, alternative features) as rectangles. The type (i.e., common, optional, alternative) is attached to rectangles and rounded rectangles as a stereotype in angle brackets. In addition, the *min* and *max* attributes of alternative domains are shown in parentheses next to the stereotypes. Rectangles with a truncated edge represent crosscutting features.

Example 7.2

To illustrate the application of the grammar, a conceptual product line model is derived from the domain requirements captured in the example on page 106. The notation $\frac{\text{terminal}}{\text{attribute: value}}$ is used to show the value of an attribute associated to a terminal. The result can be written syntactically as follows:

$$\begin{gathered}
\frac{\text{RD}}{\text{name: “Information Systems”}} \;\Big(\; \frac{\text{cf}}{\text{name: “Introduction to IS”}} \; \frac{\text{ccf}}{\text{name: “Layout-General”}} \\
\frac{\text{CD}}{\text{name: “Databases”}} \;\Big(\; \frac{\text{cf}}{\text{name: “Introduction to DB”}} \; \frac{\text{CD}}{\text{name: “DB Modeling”}} \;\Big(\; \frac{\text{cf}}{\text{name: “ER Model”}} \\
\frac{\text{cf}}{\text{name: “Relational Model”}} \;\Big)\; \frac{\text{OD}}{\text{name: “DB Theory”}} \;\Big(\; \frac{\text{of}}{\text{name: “Normalization”}} \;\Big) \\
\frac{\text{OD}}{\text{name: “DB Implementation”}} \;\Big(\; \frac{\text{of}}{\text{name: “SQL”}} \;\Big)\Big)\; \frac{\text{OD}}{\text{name: “Workflow”}} \;\Big(\; \frac{\text{cf}}{\text{name: “Introduction to WF”}} \\
\frac{\text{AD}}{\text{name: “WF Modeling”, min:1, max:2}} \;\Big(\; \frac{\text{af}}{\text{name: “Petri Nets”}} \; \frac{\text{af}}{\text{name: “Event-driven Process Chains”}} \\
\Big)\; \frac{\text{OD}}{\text{name: “WF Implementation”}} \;\Big(\; \frac{\text{of}}{\text{name: “WFMS Architecture”}} \;\Big)\Big)\;\Big)
\end{gathered}$$

The result is depicted with the graphical notation in Fig. 7.2. The root domain *Information Systems* and the subdomains *Databases* and *Workflow*, as well as the

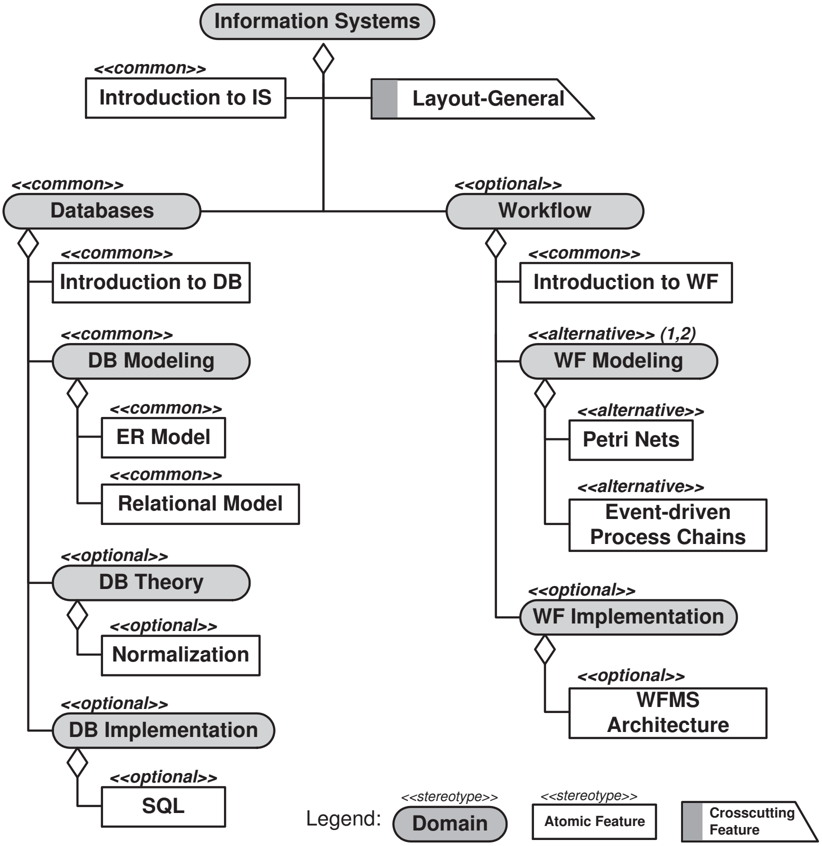


Fig. 7.2. Example for a conceptual product line model for electronic courses in information systems.

hierarchy of their subdomains are derived from the specifications of product portfolio scoping and information domain scoping. Since a course must cover databases and workflow, or only databases, *Databases* must be a common domain and *Workflow* an optional domain. The atomic features are derived from the captured requirements and they are assigned to the existing domains; common features are derived from the common requirements (e.g., *ER Model*), optional features from optional requirements (e.g., *Workflow Implementation*), and alternative features from alternative requirements (e.g., *Petri Nets*). Their types are determined from the description of the domain requirements, e.g., the requirement that database modeling must discuss ER models leads to type *common* for the feature *ER Model*. Crosscutting features are derived from the specifications of crosscutting concerns. In this case, the layout specification is a crosscutting concern for all atomic features. Thus, the crosscutting feature is included below the root domain to signalize that it will be weaved later in application engineering into all atomic features in the domain where it is located and all of its subdomains. □

Design Rationale and Expressive Power

The grammar of the conceptual product line model is based on the concepts of a feature tree introduced in Def. 4.4, however, with several extensions: domains are used to group related features or domains, features are assumed to be indivisible modules of information, features composed of other features are not allowed. One reason why composition of features out of other features is omitted in the conceptual model is that the necessary composition operator would need a precise definition of what composition means, but this definition would depend on several implementation details – a level that is closer to implementation than to the conceptual level – like for example the chosen component model (which can also vary depending on the application area). At this stage, such details are therefore kept away from the conceptual level. Instead, these details are dealt with in the construction specification on physical design (cf. Sect. 7.2.3).

The conceptual product line model is intended to be used on a general level as an initial entry point in application engineering to help application engineers choose a content configuration, and in domain engineering as a planning instrument for reuse. The model implicitly determines a context for content reuse. For example, a content developer can deduce in domain engineering from the model that when certain content is covered in a certain information product (e.g. "ER model"), some other content must also be covered (e.g., "Introduction to DB"), and create the content components accordingly. Details on the usage of the conceptual product line model in application engineering are given in Chapter 8.

The grammar is designed in such a way that circular relationships between domains and features are not allowed. Of course, the models could have been designed to have more constructs with additional semantics or sequencing information. However, a too meticulous specification with too many details has the disadvantage that the model becomes too difficult to understand as it looses its ability to abstract, and changes coming later throughout the evolution of the product line may be too difficult or tedious to incorporate, which could drastically reduce the benefits of such a model. Even if tool support would partly ease the handling of models with complex interrelationships, it still remains necessary for designers to be able to understand the models, and this can be influenced and made easier with the allowed constructs of a model. In addition, it has been realized in other contexts that "different product-line managers might simply ignore a structure that is too elaborate"[4] [MZ96].

The current restrictions of the conceptual product line model stimulate standardization of products at an early phase, and will be rewarded in application engineering. Another advantage of the choice of the model constructs will become evident in the context of the construction workflow model, which can use patterns for the creation of common, optional, or alternative features

[4] This is another reason for not incorporating a creation sequence of features yet in this model.

(see Sect. 7.2.3). As far as the ease of use is concerned, it is considered that the constructs and rules of the grammar are quite intuitive and easy to understand without technical details, which can be helpful in discussions with customers.

With the proposed grammar, PLANT attempts to find a balance between expressiveness and ease of use. As can be observed in Fig. 7.1, the grammar is context-free because on the left side of each production P there is only one non-terminal. However, the additional constraint that a terminal with one particular value of its name attribute must not occur more than once, makes the language context-sensitive.

The Content Component Model

In PLANT, a *content component* is understood to be a unit of content that can be used for composition in information products; this unit of content has a specified structure and can be deployed independently and can be subject to composition by third parties (see also [SGM02]). A content component is used to implement one atomic feature of the conceptual product line model. On a conceptual level, the *content component model* should

- be uniform for all content components in the product line;
- specify uniformly for all components: structure/encoding, metadata, way of presentation, delivery format;
- be applicable in the product-line context, i.e., allow a composition of content components to digital information products.

In PLANT, a content component model is defined in advance, i.e., during the initiation of the product line, and every content component in the product line will adhere to the same component model. This is to make sure that content components will indeed “fit” together when an information product is created later in application engineering.

Although explicit dependencies between content components are not modeled in PLANT (i.e., no explicit links or other similar constructs), the context of reuse of a content component can be derived through the associated feature in the conceptual product line model. Contrary to an unplanned reuse of content, it is here an advantage for the developer to know in advance in which product configurations the developed content can occur, and create the content accordingly.

The information relevant for the choice or design of a content component model is obtained from asset scoping (Sect. 7.1.1). However, the PLANT approach does not impose a specific content component model. Depending on the specifications in the domain requirements documents, a chosen content component model may range from a more simple one to a more complex one. For example, a component model can be even based on HTML specifications for a page with a standardized structure, metadata, and layout for the contained text, pictures, or videos. For digital information products in the e-learning domain, a suitable content model can be chosen from already existing ones (see Sect. 3.3.1). For other areas (e.g., electronic books or newspapers) a specific component model can be defined, for example using XML DTDs [HM04] for the structure and XSLT/XSLT-FO [HM04] for presentation. Alternatively, existing formats such as DocBook [Doc07] can be used.

7.2.2 Logical Design: Preparing Configuration Management

A uniform versioning scheme is defined for core assets, called the *core asset version graph*. It defines the available versions of core assets that can be used in different configurations. PLANT imposes the usage of core asset version graphs especially for content components and for the artifacts of crosscutting features; if it is necessary in a particular context, other core assets may be versioned as well using this scheme.

In addition, a *product map template* is prepared for the usage in application engineering, to capture which versions of core assets are used in a specific information product. In principle, the product map adapts the concept of a baseline to the context of digital information products (cf. Sect. 6.3).

The Core Asset Version Graph

PLANT uses a standardized versioning scheme for all core assets: the core asset version graph. It is defined as a directed, acyclic graph in which one node is considered to be an initial version (and an entry point). A version is represented as a node in the graph, and changes between versions as arcs, which result in other versions. A node may have additional attributes, such as the file path of the file (i.e., artifact) that realizes that particular version, the file type, textual comments, etc. In PLANT, exactly one core asset version graph must be associated to every atomic and crosscutting feature in the conceptual product line model. Semantically, this means that one particular feature can be realized later in application engineering only by choosing one of the available versions from its associated core asset version graph. For content components, each version (i.e., node) in the graph must be adhere to the content component model.

The graph is in fact a combination of a state-based and a change-based version model [CW98, PV05]. Although one of the two models would theoretically suffice, from a creator's point of view it is additionally helpful to describe what was changed between different versions. Even though this might be automatically computed as the difference between two versions, the integration of additional metadata (for example with the rationale behind the changes of content) into a state-based model is sensible from a practical point of view.

The application of the core asset version graph in PLANT is illustrated as an example in Fig. 7.3. The feature "Petri Nets" can be realized only by the versions of its associated version graph. A node, which represents a version, provides links (e.g., via the file path attribute) to the concrete digital artifact that makes up a version. In case that more than one artifact belongs to one version, it is assumed that there is one "header" artifact that is linked in the node. For example, in a content component model based on Web pages which may consist of text, pictures, and applets, exactly one HTML file is denoted as the "header". The artifacts which are referenced from the nodes of the version graph must either exist in a predefined location, or they will be created in domain realization. The labels of the arcs between to version nodes are in fact identifiers referencing more detailed descriptions of the changes.

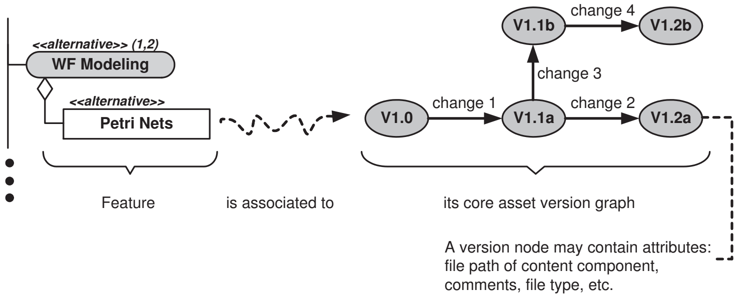


Fig. 7.3. Example for an association between a feature in the conceptual product line model and a corresponding core asset version graph.

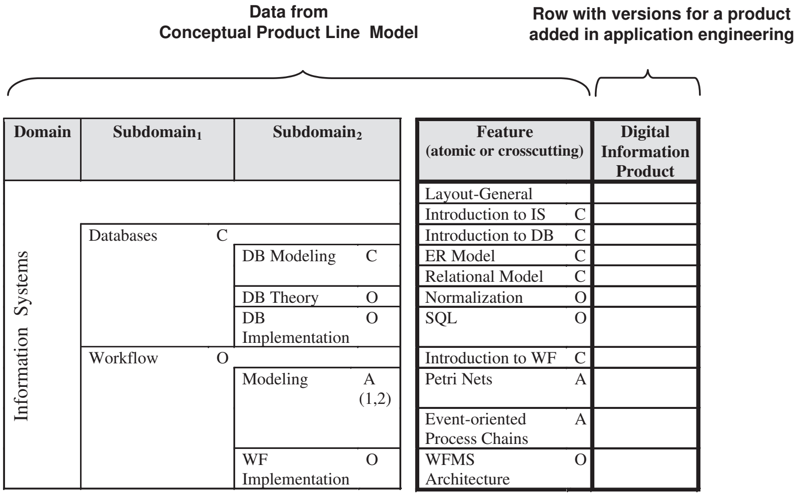
Core asset version graphs are useful to handle the evolution of core assets throughout the lifecycle of information products. The product map template, which is presented next, helps to relate the existing versions to the features of a specific information product.

The Product Map Template

The product map template is a means for a product to manage the configurations of employed core assets in systematic way. This template is filled with data obtained from the conceptual product line model and with data from application engineering.

From a general point of view, a product map template is a two-dimensional matrix as shown in Fig. 7.4, listing all available features along one dimension, and a particular digital information product along the other. The available features are obtained from the conceptual product line model. Optionally, the corresponding domains and their types can be noted on the left side to visualize the existing constraints.

In application engineering, this template is used to record the configuration chosen for one product. This is done by inserting in a cell at an intersection of a row (feature) and a column (digital information product) a version chosen from the core asset version graph that is associated to the particular feature. Such an insertion can only be made if it is allowed by the constraints of the conceptual product line model.


Data from
Conceptual Product Line Model
Row with versions for a product
added in application engineering

Domain	Subdomain$_1$		Subdomain$_2$		Feature (atomic or crosscutting)		Digital Information Product
Information Systems					Layout-General		
					Introduction to IS	C	
	Databases	C			Introduction to DB	C	
			DB Modeling	C	ER Model	C	
					Relational Model	C	
			DB Theory	O	Normalization	O	
			DB Implementation	O	SQL	O	
	Workflow	O			Introduction to WF	C	
			Modeling	A (1,2)	Petri Nets	A	
					Event-oriented Process Chains	A	
			WF Implementation	O	WFMS Architecture	O	

Fig. 7.4. A product map template from a general point of view.

From a technical point of view, the product map is realized in PLANT as a relational database that is associated to one product line. In this context, the aforementioned product map template corresponds to a database schema. The usage of a database has several advantages: redundant entries can be avoided; the data can be queried with SQL and analyzed in various ways, especially during the assembly of an information product; the constraints where and when it is allowed to make entries can be expressed as intra- and inter-relational constraints; it can be easily extended to hold additional data.

An Entity-Relationship (ER) model for this schema is depicted in Fig. 7.5 using the (min,max) notation for cardinalities and showing primary key attributes underlined; for n-way relationship types with $n > 2$, the 1:n notation is used as in [SS83] to express additional semantics. The main rationale behind the entity types and relationship types is briefly discussed, starting in the upper right corner and continuing clockwise. The entity types *Domain* ($E1$), *Feature* ($E2$) hold entities of the conceptual product line model, and $R1$, $R2$ model the containedness of domains and features. As explained, a particular feature is associated via $R3$ to exactly one *Core Asset Version Graph*, realized by $E3$ – $E5$, and $R4, R5$. A node of a graph is represented by an entity of type $E4$. The data related to a change from one version node to another is stored in $E5$. $R5$ records which versions evolved from which other versions with which changes. The *packages* ($E6$) are obtained from the packaging requirements and are used to denote bundles of core assets for delivery. The data for $E1$–$E6$ and $R1$–$R5$ is obtained from domain engineering (conceptual product line model, core asset version graphs, packaging requirements).

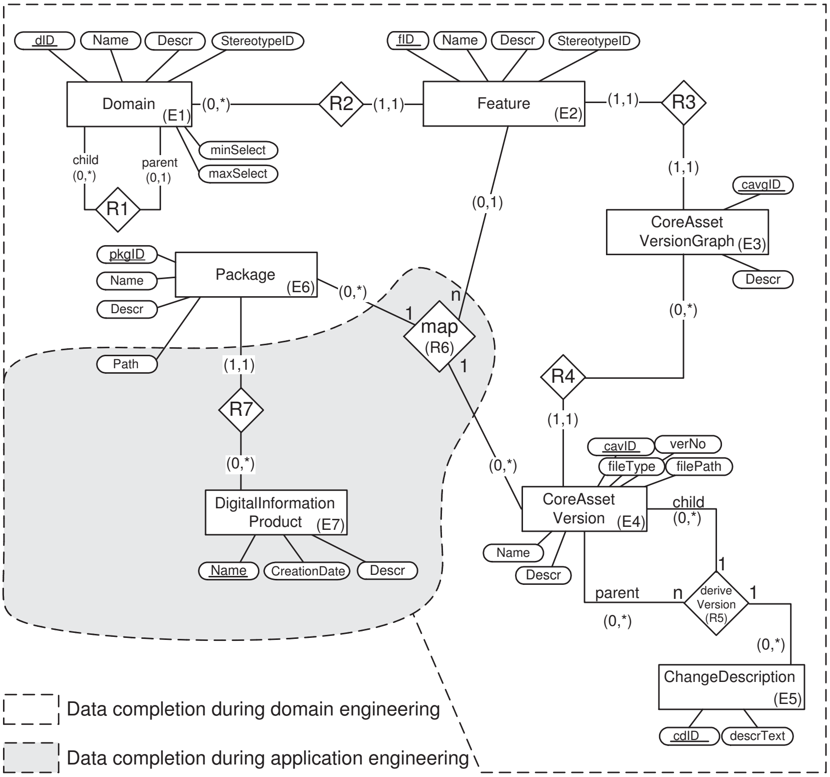


Fig. 7.5. Data model for a product map template.

The data for $E7$, $R6$–$R7$, and for some attributes of $E6$, is known only later in application engineering, and will be completed at that time. $R6$ captures which features are realized in a package using which core asset versions. Figure 7.5 shows in white and gray shades when the respective data will be completed.

7.2.3 Physical Design: Defining the Construction Workflow Model

So far, the conceptual product line model and the product map template are only capable to describe what products can be eventually built, but not how they are built, in which order the features should be incorporated, or how the content should be packaged. These specifications are more implementation-dependent and thus delayed until physical design. Here, a workflow model is designed for the product line as a whole, which means that each of its digital information products can be created when this workflow model is executed.

The workflow model specifies the tasks of the construction process and defines the sequences or options of how features can be incorporated into products. PLANT assumes that the tasks in the workflow model call helper programs with some parameters, and that a feature of the conceptual product line model is realized by one task. The definition of such a workflow model for the hole product line guarantees that digital information products can be built only in a predefined way. In principle, the construction of a specific product (later in application engineering) corresponds to a specific path taken in the workflow model, and a product variant can be specified precisely by enumerating the respective tasks; more details are presented next.

The Workflow Model

The workflow model is basically the counterpart of the construction specification C (cf. Def. 4.4) defined for software product lines. It is defined in PLANT using *$\underline{Q}$uery and Program E$\underline{x}$ecution Nets (QX nets)* which can manipulate and extract data from a relational database, and which can call helper programs with textual parameters obtained from the database. In the following, it is assumed that the underlying database is that which was designed in logical design (see Sect. 7.2.2). QX nets are an extension of *workflow nets* [AH02a] which, in turn, are *Petri nets* [Pet62] with some additional constraints. The definitions of Petri nets, workflow nets, and QX nets are introduced subsequently.

Petri Nets

Definition 7.1 (Petri Net). *A* Petri net $N = (P, T, F)$ *is a directed bipartite graph consisting of a finite set of* places P*, a finite set of* transitions T *with* $P \cap T = \emptyset$*,* $P \cup T \neq \emptyset$*, and a set of directed arcs going from places to transitions or transitions to places, represented by the* flow relation $F \subseteq (P \times T) \cup (T \times P)$*.*

Petri nets have two types of nodes, places and transitions (both are also called net elements), and it is allowed to connect only places to transitions or transitions to places. In a graphical notation, places are drawn as circles, transitions as rectangular boxes, and the flow relation as directed arrows between elements. A place p is called an *input place* of a transition t if and only if there is an arrow from p to t; a place p is called an output place of t if and only if there is an arrow from t to p [AH02a]. As a further notational convention, $\bullet t$ denotes the set of input places or *pre-set* of a transition t. Similarly, $t\bullet$, are called the *output places* or *post-set* of t. The sets $\bullet p$ and $p\bullet$ are defined analogously for places.

Definition 7.2 (Marked Petri Net). *A marked Petri net* (N, W, M) *consists of a Petri net* $N = (P, T, F)$*, a weight function* $W : F \rightarrow \mathbb{N}$*, and the marking* M *that is a mapping* $M : P \rightarrow \mathbb{N} \cup \{0\}$*, which is represented by the vector* $M = (M(p_1), \ldots, M(p_n)) \in \mathbb{N_0}^n$*, with* $n = |P|$*.*

Beyond their structural aspect, Petri nets have a dynamic aspect that comes with the tokens that places can contain. A marking defines a state of the Petri net. A *marking* M is a vector which shows the number of tokens assigned to each place of the net. A Petri net N with an initial marking M_0 is denoted (N, W, M_0). In a graphical notation, tokens are drawn as black dots inside places; if there is more than one token inside a place, the number of tokens can be written inside the place instead of the corresponding number of black dots. In a marked Petri net, a weight number $\geqslant 1$ can be associated with an arrow; the number appears besides an arrow if the weight is greater than 1.

Definition 7.3 (Occurrence Rule). *Given a marked Petri net* (N, W, M)*. A transition* t *is* enabled *under* M *if in every place* p *in* $\bullet t$ *there are at least as many tokens as the weight* $W(p, t)$ *denotes, i.e.,* $M(p) \geq W(p, t)$*. An enabled transition may* occur*, i.e., it decreases the number of tokens in each* p *in* $\bullet t$ *by* $W(p, t)$ *and increases the number of tokens in each place* p' *in* $t\bullet$ *by* $W(t, p')$*.*

It is assumed that places are passive elements that can hold tokens, and that transitions are active elements that may change the number of tokens in places, according to the fixed *occurrence rule*, as introduced in Def. 7.3. In this context, transitions can be interpreted as workflow tasks that perform some work, and places as repositories to store tokens that can symbolize signals or events (other extensions are shown in [OS96] for different types of nets). The term *workflow instance* denotes a copy of the workflow graph (i.e., Petri net) created for execution, together with an initial marking and other sequel markings obtained with the occurrence rule during execution.

Workflow Nets

In the sequel, a workflow net [AH02a] will be used to model the control flow of the construction process for digital information products.

Definition 7.4 (Workflow Net (WF-net)). *A Petri net* $N = (P, T, F)$ *is a WF-net if and only if:*

- *There is a source place* $i \in P$ *such that* $\bullet i = \emptyset$*;*
- *there is a sink place* $o \in P$ *such that* $o\bullet = \emptyset$*; and*
- *every node* $x \in P \cup T$ *is on a directed path from* i *to* o*.*

The workflow net imposes additional constraints on a Petri net. It follows from Def. 7.4 that there must be exactly one source place i and exactly one sink place o, i.e., the corresponding control flow must have exactly one start place that does not have arrows pointing to it, and one defined end with a place that does not have outgoing arrows. Furthermore, no "dangling" transitions are allowed, i.e., every transition (activity) must be on a path between start and sink.

Query and Program Execution Nets (QX nets)

QX nets are used to model the construction workflow in PLANT, and are defined in the following way:

Definition 7.5 (Query and Program Execution Net (QX net)). *A QX net (P, T, F, Progs, Params, DB, Q) is a WF-net (P, T, F) with the following extensions:*

- $Progs := \{p_1, \ldots, p_k\}$ *is a set of executable programs, called helper programs*
- $Params$ *is a set of strings representing parameters for programs* $p \in Progs$. *For* $par_1, par_2 \in Params$ *the notation* par_1, par_2 *is used to denote the concatenation.*
- $DB := (R|\Sigma_R)$ *is a relational database*[5] *consisting of a set of relations* $R = \{r_i : rel(A_i|\Sigma_i)|i = 1 \ldots v\}$ *with* $rel(A_i|\Sigma_i)$ *denoting the relation over an attribute set* A_i. *The set* Σ_i *specifies intrarelational integrity constraints over* A_i; Σ_R *specifies interrelational integrity constraints over* R.
- $Q = QM \cup QS$ *is a set of SQL query strings representing queries which can be executed on* R *in* DB, *where*
  - *the queries* $QM := \{qm_1, \ldots, qm_j\}$ *are update queries. They are assumed to modify the database state from one consistent state into another (e.g., using operations like insert, update, delete); for* $qm \in QM$, *the notation* $exec(qm)$ *is used to denote the execution of* q *in* DB
  - *the queries* $QS := \{qs_1, \ldots, qs_n\}$ *produce a string as a result. For* $qs \in QS$, *the notation* $exec(qs)$ *is used to denote the resulting string after the execution of* qs.
- *Every transition* $t \in T$ *has the form* $t = (tid, q_{pre}, q_{post}, p, params)$ *where*
  - tid *is a unique ID of the transition*
  - $q_{pre}, q_{post} \in QM \cup \{\lambda\}$ *are SQL query strings and* λ *is the empty string.*
  - $p \in Progs \cup \{\lambda\}$, *where* p *is an executable program or an empty operation*
  - $params = par_1, \ldots, par_k, exec(par_{k+1}), \ldots, exec(par_l)$ *with* $par_1, \ldots, par_k \in Params \cup \{\lambda\}$ *and* $par_{k+1}, \ldots, par_l \in QS \cup \{\lambda\}$ *i.e., params is a string with parameters for the execution of program* p, *which originate from the set of predefined parameters or from values retrieved from the database DB.*

[5] This definition of a relational database is based on [SS83].

The occurrence rule for QX nets is as follows: a transition t is enabled under the same circumstances as in Def. 7.3. However, during the occurrence of t additional steps are executed sequentially:
begin atomic

1. *the number of tokens in all input places* p_i *is decreased by* $W(p_i, t)$
2. $exec(q_{pre})$
3. *execute program p with its respective parameters params*
4. $exec(q_{post})$
5. *the number of tokens in all output places* p_j *is increased by* $W(t, p_j)$

end atomic

The main idea of QX nets is that workflow nets are extended in such a way that transitions (that represent tasks to perform construction work) can call external programs with some parameters when they occur. The parameters can be, if defined, fixed strings or strings that are extracted with SQL queries from the relational database designed in logical design. In addition, data in the database can be updated throughout the execution. Thus, configuration data for different digital information products can be stored, modified, or updated more efficiently in a relational database, and at the same time this data can as well be used in the product construction process.

For a net with a given initial marking, a variant of a product can be described by a sequence of transitions which leads to the (unique) final marking. It is assumed that a feature in the conceptual product line model is realized by one transition of the QX net[6]. This will suffice in most situations. If more than one transition is needed to realize a feature, the concept of refinement of a transition can be used, which is similar to the concept of a procedure call in programming languages. Using refinement, the transition that is used to realize a feature, will be replaced before execution by an associated transition-bordered subnet (see [Bau96] for details). This concept is widely known and applied for Petri nets, and can be used in a similar way for QX nets. For these reasons, further details are omitted here (cf. [DO96, PS05a]).

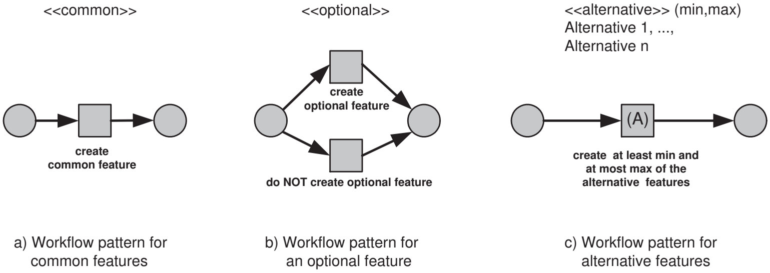
The tool that executes the workflow model (cf. Chapter 9) is assumed to make sure that the steps during an occurrence of a transition are executed in an atomic way, i.e., similar to database transactions [WV01], either all of them are executed successfully or none of them is executed. If any step fails, then the previous consistent state of the database and that of the QX net are restored. For example, the steps (2) and (4) can fail because an update query may violate some constraints defined in the database and thus cannot be successfully executed; this error is reported by the underlying database management system. Step (3) can fail when a program cannot be executed; this error can be caught with wrappers built around a program. To guarantee

[6] Note that the converse it not true, as there might be transitions that do not have a feature counterpart in the conceptual product line model, e.g., for control flow regulation, etc.

atomicity, the changes in the database should only be committed after step (5). The reversal of an aborted or failed program execution, however, is more difficult, and there are several ways to technically deal with this. For example, before the actual execution, the program with all inputs is first executed in an identical copy of the specific execution environment to check if the execution can be successful. Another possibility is the specification of a separate compensation workflow that reverses unwanted results in case of failure. Finally, it is important to mention that if there are no failures during the occurrence of any transition of a QX net, this net behaves as a classical Petri net in Def. 7.3.

A Graphical Notation for QX nets

For the graphical notation of QX nets, the aforementioned notation of Petri nets is used. In addition, the stereotypes $<< common >>$, $<< optional >>$, $<< alternative >>$ are introduced as comments that do not influence the behavior of the net. The programs that are called inside transitions, and internal SQL statements are not depicted in order to avoid cluttering; if needed, they can be added on separate sheets. Figure 7.6 shows typical workflow patterns that are frequently needed. For the creation of an alternative feature, a shorthand notation is used (called an *alternative transition*) which means that inside this transition there are *max* transitions ($min, max \in \mathbb{N}, min \leq max$), and at least *min* and at most *max* of them have to occur when the alternative transition occurs. The respective internal transitions are also omitted and can be specified on different sheets, however, the labels of the internal transitions are depicted as a comment besides the stereotype, along with the *min* and *max* numbers. In principle, an alternative transition is a shorthand notation and can be expressed by several places and transitions.



a) Workflow pattern for common features

b) Workflow pattern for an optional feature

c) Workflow pattern for alternative features

Fig. 7.6. Some basic workflow patterns used in QX nets.

Example 7.3

As an example, a workflow model (i.e., a marked QX net) is shown in Fig. 7.7 that constructs digital information products with feature configurations as in the conceptual product line model in Fig. 7.2.

It is assumed here that a digital information product will finally be realized as a file in Powerpoint format. Furthermore, it is known that the content components which it is assembled of are also files in Powerpoint format, and that there is a helper program "pptAppend.exe" that can append two given Powerpoint files sequentially one after another and produce a new file as a result, and a helper program "pptApplytemplateAll.exe" that can apply a given layout template on all Powerpoint files in a directory.

The control flow models which features can be chosen depending on the already chosen ones, as well as the sequence in which they are incorporated into a digital information product. This sequence is obtained from the packaging requirements. Each feature is created here using one transition.

As can be seen, the common feature *Introduction to IS* and the common domain *Databases* with the common features *Introduction to DB*, and *ER Model, Relational Model* (as part of the common subdomain *DB Modeling*) are created for every product, which is expressed in the sequential structure of the control flow. Then, for *Normalization* and *SQL*, which are optional features, the control flow allows to possibly override their integration into a product (e.g., the transition *do not create SQL* calls no program). Inside these transitions, it is specified that the chosen content components will be stored in the database package. Thereafter, the control flow allows to override optional domain *Workflow* and its features, and finally, weave in the crosscutting features by calling the helper program "pptApplytemplateAll.exe".

Inside one transition, e.g., *create ER Model*, basically the following things are defined: 1) a pre-execution query that can add into a database relation a new tuple that this feature was selected; 2) the helper program "pptAppend.exe" to be called with the parameters a) "c:\product1\databases.ppt" (the package where the content is to be added) and b) "c:\coreassets\ER-Ver1.ppt" (the content component used to implement the feature). The parameter b) is obtained from the database using a variable (see Chapter 9) which is bound during occurrence of the transition to the result of the SQL query "*SELECT FilePath FROM CoreAssetVersion WHERE cavID=4*"; however, this is only a preliminary value and the final definition of which version is used to implement the feature is postponed to application engineering; 3) a post-execution query adds a tuple in a relation of the database that the artifact was successfully included. □

Differences From Traditional Workflow Modeling

The approach taken in PLANT partly differs from traditional workflow modeling approaches. The latter target collaboration traditionally "in the large" with many people and require the definition of role models that describe organizational details or restrictions on data access [LR99]. This approach would be too "heavy weighted" in for PLANT, as it would require too many details. By contrast, the scope in PLANT is much smaller and the focus is less on

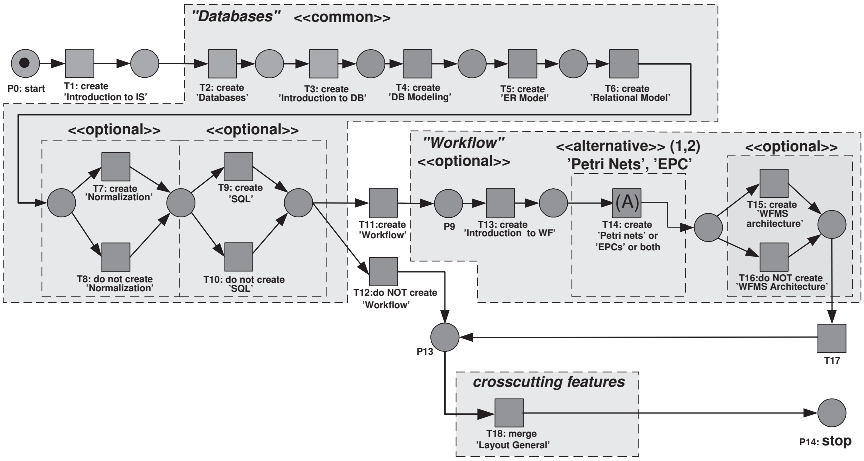

"Databases" <<common>>
P0: start
T1: create 'Introduction to IS'
T2: create 'Databases'
T3: create 'Introduction to DB'
T4: create 'DB Modeling'
T5: create 'ER Model'
T6: create 'Relational Model'
<<optional>>
<<optional>>
T7: create 'Normalization'
T8: do not create 'Normalization'
T9: create 'SQL'
T10: do not create 'SQL'
T11:create 'Workflow'
T12:do NOT create 'Workflow'
"Workflow" <<optional>>
P9
T13: create 'Introduction to WF'
<<alternative>> (1,2) 'Petri Nets', 'EPC'
(A)
T14: create 'Petri nets' or 'EPCs' or both
<<optional>>
T15: create 'WFMS architecture'
T16:do NOT create 'WFMS Architecture'
T17
P13
crosscutting features
T18: merge 'Layout General'
P14: stop

Fig. 7.7. Example for a construction workflow model.

collaboration, but more on an "agile", structured and repeatable creation of digital information products. Therefore, PLANT omits details such as role models, and assumes that the workflow can be executed on the computer of a single developer. The details of an associated tool, the Desktop Workflow Engine (DWE), which can handle workflow modeling, execution, and reuse is discussed in Chapter 9.

Reusable Workflow Modules

In the context of software product lines, the reuse of process models for the construction of products is still in its infancy and is often neglected in the literature. The need to reuse process assets has been also recognized in [GSC$^+$04, Rom05]. By contrast, the PLANT approach offers an explicit strategy and the appropriate tools for the reuse of workflow modules.

In particular, a *workflow warehouse* serves as a repository of prefabricated workflow modules (cf. Fig. 7.8) which can be reused. These modules are parts of QX nets that may represent abstract patterns (similar to workflow patterns [AH02b]) or frequently used workflows that solve a particular problem, such as format conversion, addition of metadata, attachment of a digital artifact after another one, etc. Moreover, the workflow modules are tested out and verified before being deposited in the workflow warehouse, to make sure that they operate correctly. The usage of workflow modules is intended to speed up the development of the construction workflow model, reduce manual adaptations, and make the workflow less error-prone.

The workflow warehouse is supposed to store a collection of workflow modules that are subject-oriented (i.e., for a special purpose), integrated (i.e., various developers can contribute suitable workflow modules for a certain area to

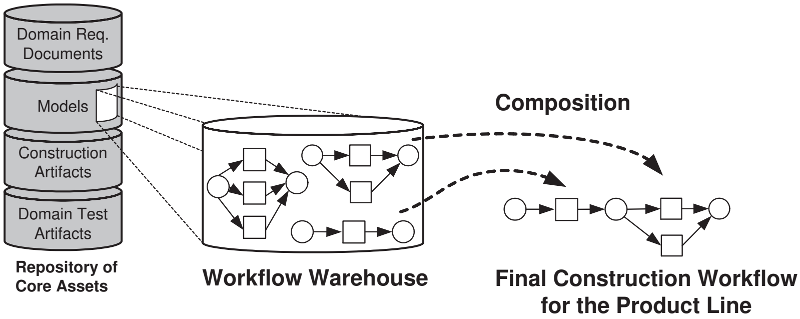


Fig. 7.8. Reuse of workflow modules kept in a workflow warehouse.

a warehouse), non-volatile (i.e., the existing workflow modules in a warehouse are not changed), and time-variant (i.e., modules are stored for a longer period of time). In this respect, there are similarities with data warehouses (see [Inm96, PS05a]).

The combination of places and transitions of copies of workflow modules with those of the edited workflow can be described in a formal way with join operators as defined in [PS05a]. Other details on the workflow warehouse and the usage of workflow modules are given in Sect. 9.2.3.

7.3 Domain Realization

In domain realization all construction artifacts are created (e.g., content components, helper programs, other digital artifacts). These artifacts are used in application engineering to construct information products. In addition, the environment is set up in which construction of a product will be done in application engineering, the product map template is implemented as a relational database, and the workflow model is implemented along with its interconnections to the database. The details of these activities are described next.

7.3.1 Realization of Content Components

All content components are created that are to be used within the product line. They are built according to the content component model (see Sect. 7.2.1). The content components can be created either by forward engineering, or by reengineering existing digital material in such a way that it adheres to the requirements of the product line.

Forward Engineering

Forward engineering means that a content component can be implemented by using the results obtained from domain engineering, such as content requirements or the content component model (see Sect. 7.2.1), in a top-down manner. Starting with the requirements for a specific content component and the uniform content component model, the content component (which contains the actual information) is created, and the finished component is stored in the repository of core assets.

Example 7.4

Some examples for content components (based on different content component models):

- A Powerpoint file of a lecture on *ER Models*
- A Web page for a lecture on *ER Models*
- An XML file of a lecture on *ER Models*
- A PDF file of a lecture on *ER Models*
- A video file of a lecture on *ER Models*

□

Reengineering from Existing Material

The reengineering approach combines a bottom-up and a top-down approach for the creation of content components. In some cases digital material is already available which could be used to create content components for a product line. However, there are obstacles in practice which sometimes impede an easy transfer, such as different file formats that possibly do not allow easy modification of content, or diverging layouts and content structures. In such cases, an adaptation of existing content to create content components that satisfy the product line requirements can be as tedious as a creation from scratch.

Reengineering can help in this context by automating some of the well-structured tasks. The principle of reengineering involves, similar to reengineering of software [CI90], the phases reverse engineering (bottom-up) and forward engineering (top-down), and is depicted in general for content artifacts in Fig. 7.9. At first, reverse engineering analyzes a given digital artifact, such as a file containing some content, and extracts some data whose representation is on a higher level of abstraction which allows for modifications. Thereafter, forward engineering creates a content component according to the content component model, which fits into the product line context, and the level of abstraction is reduced. The difference between forward engineering alone as described at the beginning of this section and reengineering, is that reengineering has additionally the phase of reverse engineering that extracts some useful data from existing artifacts, followed thereafter by forward engineering which makes adaptations.

A more detailed process model for reengineering digital artifacts, which can be used with the creation of content components, is shown in Fig. 7.10. The

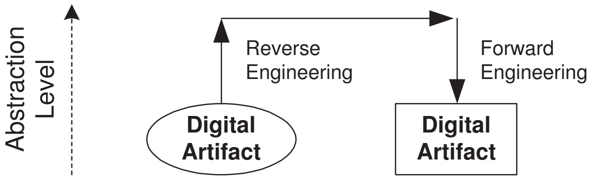


Fig. 7.9. Principle of re-engineering of existing content (see also [PSV05]).

process model assumes that from existing artifacts, some kind of components can be extracted, which can be for example, text, audio and video files, flash animations, or layout specifications. The automated component identification phase (which does reverse engineering) is performed automatically, and is followed by the semi-automatic component qualification phase for forward engineering.

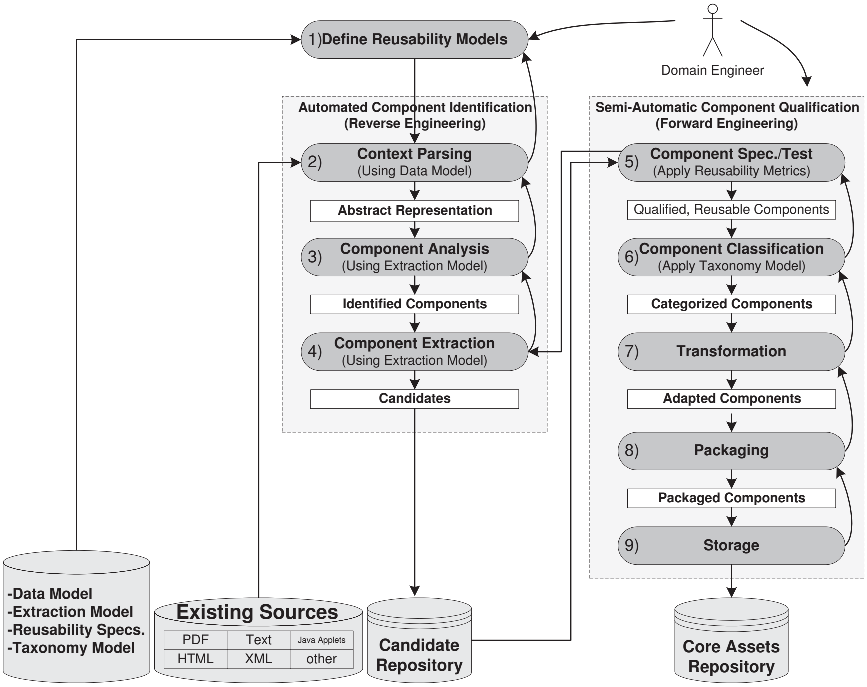

1)Define Reusability Models
Domain Engineer
Automated Component Identification (Reverse Engineering)
2) Context Parsing (Using Data Model)
Abstract Representation
3) Component Analysis (Using Extraction Model)
Identified Components
4) Component Extraction (Using Extraction Model)
Candidates
Semi-Automatic Component Qualification (Forward Engineering)
5) Component Spec./Test (Apply Reusability Metrics)
Qualified, Reusable Components
6) Component Classification (Apply Taxonomy Model)
Categorized Components
7) Transformation
Adapted Components
8) Packaging
Packaged Components
9) Storage
-Data Model
-Extraction Model
-Reusability Specs.
-Taxonomy Model
Existing Sources
PDF
Text
Java Applets
HTML
XML
other
Candidate Repository
Core Assets Repository

Fig. 7.10. A reengineering process model for content components (cf. [PV05]).

At the beginning of the process, a domain engineer has to define the reusability models that are used for automation (phase 1), such as the data model of the file which contains the data to be extracted (e.g., an XML DTD), the extraction model that specifies what to extract (e.g., an XQUERY definition), reusability specifications to compute how well a piece was extracted (e.g., syntax conformity, etc.). An optional taxonomy model can be defined to help to categorize extracted candidates more easily when many candidates are there. In phase 2, the data model is used to parse existing files and gain a more abstract representation, such as an abstract syntax tree [ALSU07]. This representation is used in component analysis (phase 3) which applies the extraction model to identify the parts of interest; these represent candidate components which are stored in phase 4 into a temporary candidate repository. In practice, it is possible that the aforementioned conceptual reverse engineering phases are not directly distinguishable when they are embedded inside reverse engineering tools (e.g., [Cif94]).

The semi-automatic work that needs interaction with the domain engineer begins in phase 5. At first, extracted candidates are automatically examined with the available reusability specifications. In addition, the domain engineer manually inspects and tests the extracted results to see if the components are useful and, if not, go back to earlier phases. When a large number of components need to be extracted and a taxonomy model was defined earlier, the appropriate categorization of components can be done in phase 6 to simplify their handling. In phase 7, extracted content components can be additionally transformed to fit within the product line context, i.e., the content, layout, structure, or granularity may be modified manually or with appropriate tools to satisfy the requirements of the content component model. In phase 8, a finished content component can be "packaged", i.e., additional metadata can be added if necessary, or the delivery format can be changed (e.g., to a compressed file format to save space). Finally, the packaged content component is stored in the core assets repository.

Reengineering is especially attractive when an organization has lost the original source files of content components, which are only available in a binary format that does not allow simple modifications, or when creation from scratch is too expensive or not possible. Reverse engineering, however, can have many intricate legal implications which can differ in various countries and license agreements; these aspects are discussed in [Sam90, Hoe06]. Additional details on the process model and its activities can be found in [PV05].

7.3.2 Realization of other Construction Artifacts

Other artifacts are possibly needed within a product line in order to be able to create products. Depending on the chosen content component model, *layout specifications* may be separately defined. If possible, such a separation should be always be done, because it makes the individualization and the maintenance of content easier. For example, Powerpoint layout specifications can be separately saved in a master file; for HTML-based content one can create separate CSS files, for XML one can create formatting guidelines with XSL-FO.

In many cases, *programs* need to be integrated *within textual content.* Such programs are also created in domain realization. For example, programs that are often embedded into content are Flash animations or Java applets (e.g., simulations, etc.).

Additional *helper programs* are created if the combination of content components and other construction artifacts cannot be performed with available tools, and especially if it is not possible to integrate the tools into an automated workflow. Helper programs need to be designed in such a way that the construction of digital information products can be made by calling these programs with certain parameters. Helper programs can be for example format converters or wrappers that provide for the workflow a standardized interface for calling different programs. Helper programs are a critical part of the automation of the construction workflow model for digital information products.

Depending on the application domain, the testing of construction artifacts may become tedious and inefficient when many different cases have to be tested. If such cases can be derived in an algorithmic way, *test case generators* can be created to simplify this task. Such generators are also reusable in application testing.

Example 7.5

Some examples of other construction artifacts:

- **Layout-specifications:** A Powerpoint master file “master.pot”
- **Helper programs:**
  - *pptappend.exe*: appends the slides of a Powerpoint file after the last slide of another Powerpoint file; accepts 3 parameters: source file 1 and 2; 3) the output file.
  - *pptapplytemplate.exe*: applies a layout file to a Powerpoint file; accepts 2 parameters: 1) a Powerpoint file; 2) a Powerpoint master file
  - *pptextracttemplate.exe*: for reengineering purposes, this program extracts a layout specification from an existing Powerpoint file; accepts 2 parameters: 1) a Powerpoint file; 2) the file name in which the extracted master is to be stored.

□

7.3.3 Realization of the Product Map Database

The product map template (see Fig. 7.5) is implemented as a relational database. In addition, the data available from domain engineering is initially entered into the appropriate tables (i.e., relations): the domains, features, core asset version graphs for the construction artifacts, package data.

7.3.4 Realization of the Construction Workflow Model

The model of the construction workflow for the product line that was designed in physical design (see Sect. 7.2.3) is realized with a tool that can handle the details of QX nets (see Chapter 9). The tool also represents the environment in which digital information products are created in application engineering. Predefined workflow modules can be reused from the workflow warehouse (Fig. 7.8).

7.4 Domain Testing

Domain testing in PLANT is concerned with the *quality assurance* of the core assets developed in domain realization. In PLANT, *testing* is understood as a process in which core assets, (esp. construction artifacts such as content components or helper programs, and the construction workflow model) are analyzed under specified conditions, and the results are observed, recorded, and evaluated. The notion of testing is broader than in traditional software development, since construction artifacts can be a mixture of content and software, and therefore poses additional difficulties compared to pure software testing techniques [Som04b, AM05, Pre05]. For parts that are pure software (e.g., Java applets, etc.) traditional testing techniques as described in [Som04b, AM05, Pre05] can be used; they are well-known and thus will not be discussed here in detail. The remainder of this section therefore focuses on information-product-specific aspects: content component testing, formal verification of the construction workflow model, and the integration testing of content components. Since content itself is typically not "executable", this kind of testing is often static.

7.4.1 Testing Content Components

Content component testing is done for all content components created in domain realization, and focuses on single components only. The test cases and other artifacts (such as reports) created in domain testing are stored in the core asset repository, so that they can be reused in application engineering.

On a syntactical level, content components are checked to make sure that their structure is as defined in the content component model. This task can be automated with syntax checkers in cases in which the content component model has a formal specification (e.g., a DTD when XML or HTML are used). In addition, it is tested whether available layout specifications can be (automatically) applied to content components, and it is validated that the results are as expected, i.e., the component has the specified layout (e.g., colors, text sizes, etc.), there are no new, unwanted text wraps or other unexpected changes. Furthermore, tests are done to make sure that the content is correctly displayed in the target environment. This is especially critical for some file formats, such as HTML, since a valid HTML file can be displayed differently by different browsers in different screen resolutions. This is because the respective standards do not specify all details of the rendering mechanisms [W3C07a]. In this context, a possible criterion against which can be tested is the degree to which the same file is displayed differently in different browsers; this degree can be evaluated by automated searches in HTML code for known incompatible constructs, or empirically by human testers according to certain standardized criteria.

Finally, a domain engineer has to validate each content component by asking the questions: is this what was really desired? Is the content contained in one component independent enough so that, when combined with other content components in different allowed configurations, the result makes sense? Does the way the content is spread over different components make sense? The testing of content components stops thereafter successfully if no errors were encountered in any of the components; otherwise, a re-iteration of earlier domain engineering phases is necessary.

7.4.2 Verifying the Construction Workflow Model

The verification of the QX net representing the construction workflow model is a check which ensures that it can really generate only products with configurations allowed by the conceptual product line model.

There are various techniques from the area of Petri nets which can be applied in this context. As already mentioned, QX nets behave as traditional Petri nets when the following assumptions hold: 1) in the analysis, alternative transitions behave like traditional transitions (i.e., it is abstracted away that they might contain other transitions; such internal transitions are ignored); 2) the correct functioning of an alternative transition is ensured by other mechanisms; 3) programs called inside transitions are executed without errors (ensured by other mechanisms); 4) queries in the database are executed without errors.

Reachability analysis can ensure that a marking representing the state that a certain feature was built can only be reachable from a marking representing that some other features were already built. This way, one can check the satisfaction of the constraints of the conceptual product line model that demand that some features can be chosen only when some features in parent nodes were chosen (see Sect. 7.2.1). This analysis can be done by creating a marking graph.

Analysis of the Marking Graph

The definition of the marking graph of a marked Petri net is based on arc-labeled directed graphs [DR98].

Definition 7.6 (Arc-Labeled Directed Graph). *An arc-labeled directed graph is given by*

- *a set V of vertices*
- *a set of labeled edges (v, l, v'), representing source vertex, label, target vertex, where $v, v' \in V$ and l is a label of some given set L.*

For a given labeled edge (v, l, v'), v' is an *immediate successor* of v. A finite or infinite sequence of labeled edges $(v, l, v_1), (v_1, l, v_2), \ldots$ is called a *path* of an arc-labeled graph. All vertices $\{v, v_1, v_2, \ldots\}$ are called successors of v.

Definition 7.7 (Marking Graph). *Let (N, W, M_0) be a marked Petri net (cf. Def. 7.2) with $N = (P, T, F)$. The marking graph of (N, W, M_0) is an* arc-labeled directed graph *with a distinguished initial vertex, and edges labeled by transitions. Furthermore,*

- *the vertices represent the reachable markings,*
- *the distinguished initial vertex represents the initial marking M_0,*
- *the labeled edges are triples (M, t, M') such that M, M' are reachable markings satisfying $M \xrightarrow{t} M'$, i.e., the occurrence of transition $t \in T$ (which is enabled under M) transforms the marking of the net from M to M' according to the occurrence rule.*

In a graphical notation, the vertices of the marking graph are shown as bullets, edges as labeled arrows between two bullets, and the initial marking is distinguished by an additional arrow pointing to it.

The reachability problem can be stated as follows: given a marked Petri net (N, W, M_0) and a marking M', it should be decided whether M' is reachable from M_0. It has been shown by [May84] that this problem is decidable for the types of Petri nets used here (see Defs. 7.1 – 7.3). Using the marking graph, one can deduce various properties of a Petri net, e.g. [DR98],

- if the marking M' represented by a vertex v' is *reachable* from M_0, represented by v_0, then there exists in the marking graph a finite directed path $(v_0, t_1, v_1), (v_1, t_2, v_2), \ldots, (v_i, t_i, v')$; by definition, a marking of a vertex v_i is reachable from itself;
- a marked Petri net is called *deadlock-free* if every reachable marking enables some transition; the net is deadlock-free if and only if its marking graph has no vertex without outgoing edge; a marking that enables no transition is called *dead*;
- a marked Petri net (P,T,F) is called *b-bounded* (or simply bounded), if all its places are b-bounded; a place $p \in P$ is b-bounded if for each reachable marking, the number of tokens it contains is $\leq b$, with $b \in \mathbb{N}$. A finite Petri net is bounded if and only if its marking graph has finitely many vertices.

As discussed in Sect. 7.2.3, QX nets have exactly one sink place that does not have outgoing arrows. This means that QX nets necessarily have exactly one marking with a deadlock because there is one dead marking when the workflow reaches a “stop” state. This insight can be used for verification purposes; if a designer encounters other dead markings (i.e., in the marking graph there is more than one node without outgoing edges) this must be a design error, and the QX net must be re-designed. In addition, the marking graph reveals the preconditions in the control flow before a certain transition can occur. By interpreting a marking as a state in which some features are created, the marking graph can be used to ensure that only those product configurations can be generated which were allowed by the conceptual product line model, by checking the appropriate edges in the marking graph.

The verification is finished successfully if it is ensured that the QX net has exactly one "stop" marking and if within the marking graph only desired state changes are possible.

Example 7.6

As an example, the marking graph is constructed for the net in Fig. 7.7 with its initial marking M_0 shown there. In the corresponding marking graph in Fig. 7.11, each vertex represents a unique marking that is reachable from M_0, and each vertex is given a unique number. For clarity, the complete marking vector of a marking is omitted. For example, $M_0 = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$, $M_1 = (0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$,
$M_{14} = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1)$.

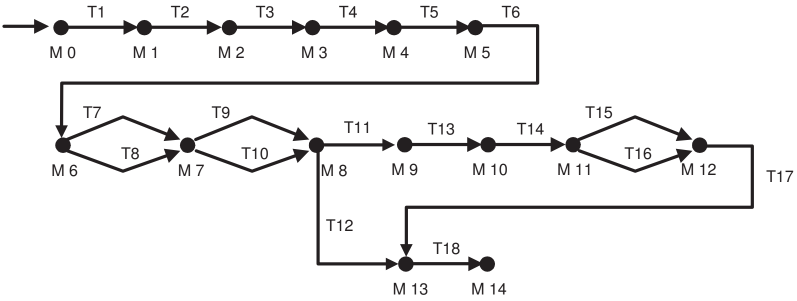


Fig. 7.11. Marking graph for the net in Fig. 7.7.

As can be seen, the final marking M_{14} (where there is one token in the place "stop") is reachable from the initial marking M_0. In addition, the net is bounded because the marking graph has finitely many vertices. Furthermore, one can deduce from the marking graph that for example the marking M_{10}, where the feature "introduction to workflow" was created by $T13$, will only be reachable if the creation of the optional workflow domain was chosen before ($T11$). □

Unfortunately, there are some drawbacks of this method which are mainly resulting from the computational complexity of Petri nets. The marking graph cannot always be computed efficiently; in terms of numbers of places, transitions, arcs, and tokens, it requires for bounded Petri nets exponential space [Lip76, May84, Sta90]. In addition, for some nets, the corresponding marking graph can have an infinite number of vertices (e.g., for unbounded nets).

However, typical construction workflow models in PLANT seem to be prevalently bounded and have a marking graph with a finite number of nodes (see also case study in Chapter 10). It becomes visible here why the possible types of constructs were limited in conceptual product line model: by allowing only common, optional, and alternative features or domains, one can use predefined workflow patterns (see Fig. 7.6) within the construction workflow model, which help to obtain nets which are likely to be bounded. In addition, marking graph analysis can be done for the reusable workflow modules stored in the workflow warehouse (see Fig. 7.8), which are usually small enough to make the computation feasible. If the marking graph approach becomes infeasible for complex nets, there are also other analysis techniques based on linear algebra, which provide semi-decision algorithms with a weaker form of reachability analysis [DR98]; other techniques are also discussed in [Sta90, DR98].

7.4.3 Integration Testing of Content Components

Integration testing in PLANT focuses on testing whether the combination of different content components works, and if a product can be created as a combination of different content components. This poses in PLANT similar problems like domain integration testing in software product lines (see Sect. 4.3.4), since after domain realization, there is not yet a finished digital information product that can be tested. In addition, there are potentially many configurations of different core assets that can be assembled to different information products, which makes an exhaustive integration testing almost impossible, or indeed infeasible.

As a tradeoff, PLANT follows therefore for the integration testing of content components the sample application strategy approach (cf. Sect. 4.3.4). This strategy creates one or more sample digital information products. For this, the product map template is complemented with sample versions for every feature in the product line. Thereafter, the construction workflow model is executed.

The generation of a sample digital information product has the drawback that only the creation of a specific feature configuration is tested, which poses the question of which and how many configurations to test. PLANT uses the marking graph of the construction workflow model (Sect. 7.4.2) to derive a finite set of products with sample configurations that are to be tested. Similar to branch coverage testing of control flow graphs of software [Bal00], the different sample products are generated here in such a way that each edge in the marking graph of the construction workflow model is passed at least once. The rationale behind this is that every state modification in the workflow is tried out at least once (an example is shown in later Sect. 10.4). Integration testing is finished successfully when all sample products are generated successfully. In case or errors, another iteration of domain engineering can be used for corrections. Another related approach is described in [DOZZ97].

As already explained in Sect. 7.4.2, the marking graph could theoretically have sometimes no finite representation or may be inefficient to compute, which would mean that integration testing could not be done in this way. However, as reasoned in the previous Section, marking graphs for the context where PLANT is used are expected to have prevalently a finite representation. This is also supported by the results of a case study carried out a "real-world" context, to be presented in Sect. 10.

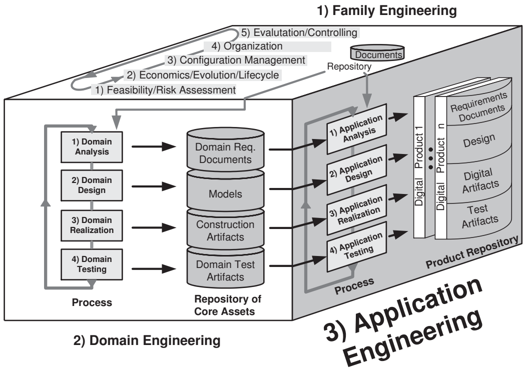
7.5 Summary and Discussion

The purpose of domain engineering in PLANT is to make all necessary preparations which are needed to generate concrete information products in application engineering. This does not only include the capturing of requirements and the creation of models (e.g., for allowed configurations, configuration management, assembly), but also the implementation of reusable construction artifacts (such as content components), and testing, validation, and verification of components or models.

Similar to software product lines, most of the effort of creating information products with PLANT has to be invested in domain engineering. However, it will become evident in application engineering how this investment and the restrictions introduced so far will pay off.

8

Application Engineering in PLANT



Application engineering focuses on the creation of a single digital information product, using the core assets prepared in domain engineering. Application analysis captures the product-specific content requirements, which ideally must be a subset of the domain requirements. Application design thereafter obtains a valid content configuration from the conceptual product line model. A product map records for the chosen configuration which versions of which core assets to use for realization. The actual generation of an information product is done in application realization. There, the construction workflow model that was defined in domain engineering is executed, and it uses the data from the product map to call helper programs during execution. The resulting information product is tested in application testing, which complements domain testing. The Chapter concludes with a summary and discussion.

8.1 Application Analysis

In PLANT, the objective of application analysis is to capture the requirements for one single digital information product that is to be created with core assets prepared in domain engineering. From the terminology of software product lines, the term "application" is kept also in this context to emphasize that a digital information product can indeed consist of a mixture of content and software.

The primary source to obtain these product-specific requirements is a customer or a group of customers who demand a certain product with some special content configuration. All other requirements are obtained from domain analysis (Sect. 7.1). In particular, application analysis collects product-specific content requirements. The output of application analysis is one or more documents that capture in natural language these specific requirements for one information product.

8.1.1 Capturing Product-Specific Content Requirements

The product-specific content requirements determine which content the considered information product should contain. Furthermore, they may specify custom additions (e.g., cross-references, etc.) to be included manually after the generation of a product out of core assets. These additions are assumed to be minor, highly product-specific and require little realization effort compared to the overall effort of application engineering. Therefore, they are not included into the models of domain engineering, and the related changes are not allowed to contradict the domain engineering models, for example by changing the content in such a way that it would no longer represent a valid configuration.

The main difference between traditional requirements engineering and application requirements engineering in PLANT is that in PLANT only those requirements can be realized, which are at the intersection of what is demanded by customers and technically feasible, and what is available in the product line. This means that in PLANT the content requirements demanded for a certain information product (except the custom additions) ideally need to match a subset of the content requirements captured in domain engineering (see Sect. 7.1.2). To ease the task of this assessment, the conceptual product line model (created in domain design, see Sect. 7.2.1) can additionally be used as a reference in application analysis[1], as it already visualizes domain requirements on a higher level of abstraction. In cases with a mismatch, an application designer must re-negotiate with a customer to achieve such a subset, or stop application engineering and hand over to domain engineering where additions can be made, such as the creation of other core assets and the modification of the models.

[1] Note that this connection is omitted in Fig. 5.1 to keep the figure clearly arranged.

Of course, for some customers not all requirements will be fulfilled. The rationale behind this approach is that on the one hand individualization is limited to a certain degree, but on the other hand, synergy effects can be realized which increase quality (because common components make updates easier and less error-prone) and reduce production costs. If negotiations with customers can only satisfy few of their requirements or if negotiations fail entirely, e.g., because there are no adequate configurations or core assets, this feedback is given to domain engineering for possible changes.

Example 8.1

Based on the example for domain requirements in Sect. 7.1.2, a special e-learning course on “Information Systems” can have the following content requirements (negotiated with a customer group):

Content requirements:

1. The course on “Information Systems” has a general introduction “Introduction to IS”.
2. The course needs the domain “databases” with “database modeling”.
3. The database domain has an introduction “Introduction to DB”.
4. The domain “database modeling” has the “ER model” and the “relational model”.
5. The course needs the domain “workflow”, and the domain “workflow modeling” with “Petri nets”.
6. The domain “workflow” has an introduction “Introduction to WF”.

□

8.2 Application Design

Using the requirements specified in application analysis, the objective of application design is to create a product-specific model and a product map, based on the conceptual product line model and core asset version graphs developed in domain engineering. The affected models and the tailoring process are explained next.

8.2.1 Deriving a Product Model

The product model determines a specific content configuration that is selected for one particular information product. In a general sense, the product model defines the “architecture” of the specific information product. Using the requirements from application design, the specific product configuration is derived from the conceptual product line model.

The product model can be regarded as an instance of the conceptual product line model. The rules that are presented in the sequel are used to derive an instance, and thus, a content configuration that is valid according to the product line constraints. From a general point of view, these represent the semantics of the grammar introduced in Fig. 7.1. There are also conceptual similarities between these rules and those presented in Def. 4.4 for software products in software product lines. The rules also unveil how synergy effects are achieved through the usage of common features or common domains.

1. A *product model PM* is a subtree of the conceptual product line model $CPLM$.
2. By definition, the *root domain* of $CPLM$ is included into PM.
3. A *domain or feature of type common* has to be included into PM if its immediate predecessor was included, i.e., for any included common domain or common feature, all immediate successors of type common have to be included in PM.
4. An *optional domain or optional feature* may be included into PM if its immediate predecessor is included.
5. If the immediate predecessor of an *alternative domain* (which has attributes min, max) was included in PM, then, the particular alternative domain must also be included. Furthermore, if an alternative domain contains only alternative features (see grammar in Fig. 7.1), at least min and at most max of these features must be chosen; if an alternative domain contains only other alternative domains, then at least min and at most max of these domains must be included.
6. A *crosscutting feature* (which has no type) has to be included in PM if its immediate predecessor was included.

Example 8.2

The application of the aforementioned rules is illustrated in this example which uses the conceptual product line model introduced in Fig. 7.2. A product model (for a product variant) should be derived that satisfies the specific content requirements of the example in Sect. 8.1.1.

As shown in Fig. 8.1, we can proceed in a top-down manner to include the appropriate domains and features from the conceptual product line model into one concrete product model; included features or domains are checked off. By rule 2, the root domain is included. Then, “Introduction to IS” is included (rule 3), “Layout-General” (rule 6), and “Databases” (rule 3). As a consequence, “Introduction to DB”, “DB Modeling”, “ER Model”, and “Relational Model” are included (rule 3). By rule 4, “Workflow” is included. As a consequence, “Introduction to WF” must be included (rule 3), and “Petri Nets” is chosen from “WF Modeling” (rule 5).

The resulting product model is depicted on the right side of the figure, and it has a valid configuration. The crosscutting feature “Layout-General” will be weaved into all atomic features of the domain where it is located, and all subdomains (see Sect.

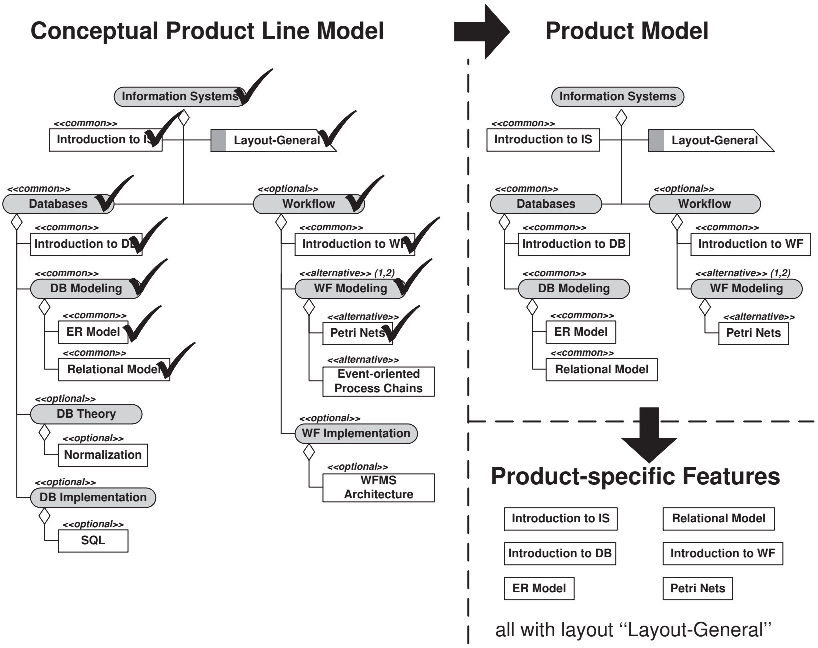


Fig. 8.1. Example of a derivation of a product model from the conceptual product line model of Fig. 7.2.

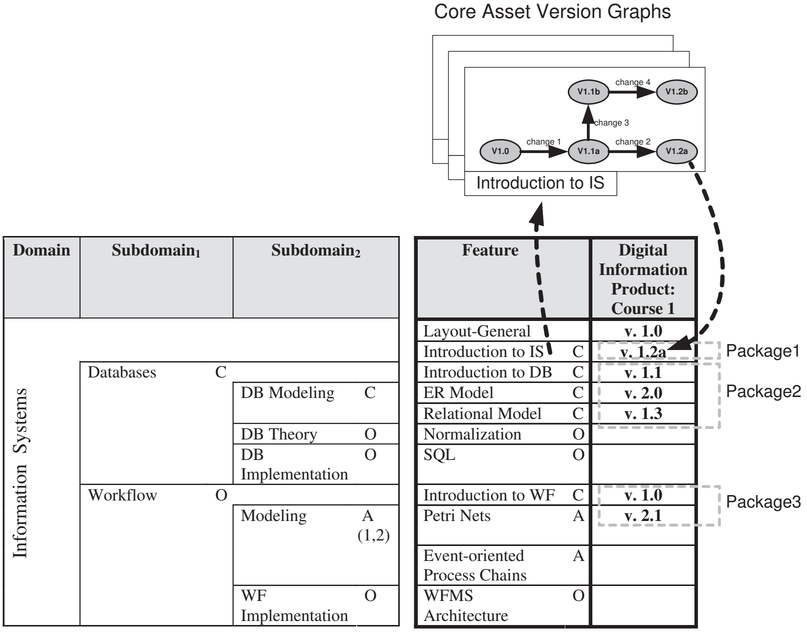
7.2). Looking at the product model, it becomes also clear which atomic features are to be realized in this particular information product (lower left side of the figure), and that their order is not yet important at this stage.

This example also clarifies that not every content configuration is allowed by the product line specifications. For example, it is not allowed to select only "SQL" and "WFMS Architecture". The conceptual product line model imposes that if these features are selected, then we must also select at least "Introduction to IS", "Layout-General", "Databases" with "Introduction to DB", and "Workflow" with "Introduction to WF". This way, a certain reuse context is guaranteed, and a content developer is aware of the possible configurations in which a certain content component is reused. □

8.2.2 Creating a Product Map

The product model specifies so far only which domains or features have been chosen for a specific information product. At this point, additional configuration details are added: each product feature is assigned a predefined version of its corresponding core asset version graph.

Essentially, the product map template that was prepared in domain design (see Fig. 7.4) is now used to do this assignment. After the data for the particular product has been completed in the product map template, it is referred to as the product map (of the particular product). The principle of how to assign a version to a feature is shown conceptually in Fig. 8.2.



Domain	Subdomain$_1$	Subdomain$_2$	Feature	Digital Information Product: Course 1
Information Systems			Layout-General	v. 1.0
			Introduction to IS C	v. 1.2a
	Databases C		Introduction to DB C	v. 1.1
		DB Modeling C	ER Model C	v. 2.0
			Relational Model C	v. 1.3
		DB Theory O	Normalization O	
		DB Implementation O	SQL O	
	Workflow O		Introduction to WF C	v. 1.0
		Modeling A (1,2)	Petri Nets A	v. 2.1
			Event-oriented Process Chains A	
		WF Implementation O	WFMS Architecture O	

Fig. 8.2. Example of a product map for a specific product.

For each feature (atomic or crosscutting) that is contained in the product model, exactly one version from its associated core asset version graph must be chosen, which is written at the intersection of the row of the respective feature and the column of the product. This assignment means that a feature has be to realized by that particular version. This assignment is similar to the built-from-relation discussed in Def. 4.4 for software product lines. Furthermore, it is indicated in the Figure how the atomic features will be grouped into delivery packages, according to the packaging requirements captured in domain engineering. Technically, the data of the aforementioned assignments is added to the product map database that was prepared in domain engineering (Fig. 7.5).

8.3 Application Realization

Application realization creates the digital information product in a generative manner with the feature configuration specified so far. This is done by an application engineer who is executing the pre-defined construction workflow model specified in domain engineering, using the tool introduced in Chapter 9. In particular, this workflow also specifies, in a pre-defined way, the order in which features are integrated into the product, and uses the data of the product map that was finalized in application design. The application engineer is not allowed to modify the structure of the net.

8.3.1 Executing the Construction Workflow Model

The execution of the construction workflow model, which is a QX net (Sect. 7.2.3), is handled in the following way. For a given marking:

1. if no transition is enabled, the execution stops. In a well-designed, marked net there must be exactly one marking with this property (when the workflow reaches the "stop" state).
2. if exactly one transition is enabled, it occurs automatically without user interaction.
3. if more than one transition are simultaneously enabled, then the application engineer must select exactly one of them to occur.

With these semantics, it becomes obvious how choices are made during the construction of an information product. Optional and alternative features or domains are modeled in a way, e.g., using patterns as in Fig. 7.6, such that more than one transition will be enabled when the control flow reaches the point where they are to be constructed. At that point, the application engineer is asked to make a choice. This choice will depend upon the product model derived in application design[2]. Generally speaking, the creation of different variants of information products depends on the different choices that can be made when more than one transition is enabled (recall that is was assumed that one feature is created by one transition, see Sect. 7.2.3). Furthermore, for the creation of different variants, different "paths" with different transitions are taken through the workflow model during execution. However, the workflow model is defined in such a way that only those paths can be taken that construct configurations allowed by the conceptual product line model.

When the execution of the construction workflow is finished, its result are the artifacts of the respective digital information product; the chosen content components are contained within the appropriate packages.

Thereafter, an application engineer can perform minor manual adaptations which do not violate the constraints of domain engineering. For example,

[2] Figuratively, the "marking off" of features or domains in Fig. 8.1 corresponds here to the choice which transition should occur.

such adaptations can be the insertion of individual cross-references inside the content, completion of metadata, etc. Finally, the realized configuration is reported to family engineering (cf. Sect. 6.3.2).

Example 8.3

As an example, the QX net shown in Fig. 7.7 is reconsidered. A figurative path is shown in Fig. 8.3, which creates a concrete information product with the features selected in Fig. 8.1.

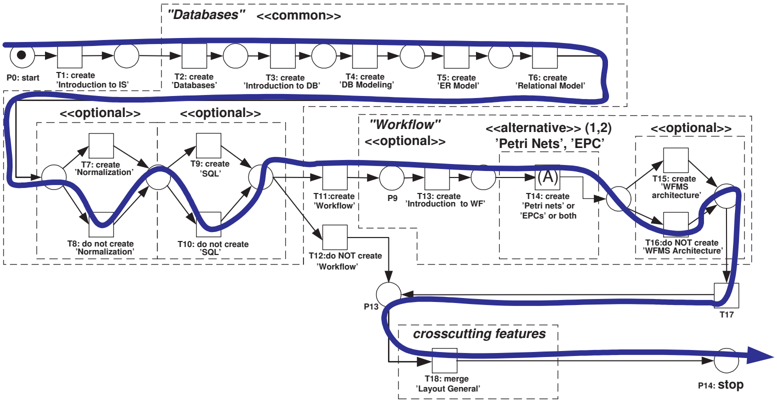

"Databases" <<common>>
P0: start
T1: create 'Introduction to IS'
T2: create 'Databases'
T3: create 'Introduction to DB'
T4: create 'DB Modeling'
T5: create 'ER Model'
T6: create 'Relational Model'
<<optional>>
<<optional>>
T7: create 'Normalization'
T8: do not create 'Normalization'
T9: create 'SQL'
T10: do not create 'SQL'
"Workflow" <<optional>>
<<alternative>> (1,2) 'Petri Nets', 'EPC'
<<optional>>
T11:create 'Workflow'
P9
T13: create 'Introduction to WF'
(A)
T14: create 'Petri nets' or 'EPCs' or both
T15: create 'WFMS architecture'
T16:do NOT create 'WFMS Architecture'
T12:do NOT create 'Workflow'
P13
T17
crosscutting features
T18: merge 'Layout General'
P14: stop

Fig. 8.3. Example of a path taken in the workflow model during execution.

In particular, one can see that the transitions occurred in the following order: $T1, \ldots, T6, T8, T10, T11, T13, T14, T16, T17, T18$. Inside these transitions, the predefined helper programs are called with parameters obtained from the product map, and create the artifacts of the information product. For the depicted path, most transitions occur automatically. The transitions chosen by the application engineer to occur are: T8, T10, T11, "Petri nets" inside T14 (recall that an alternative transition is shorthand notation, see Sect. 7.2.3), and T16. The result is a course variant of Information Systems. □

8.4 Application Testing

Application testing is concerned in a general sense with the quality assurance of the concrete artifacts of the generated digital information product. Although a part of this task has already been done from a more general perspective in domain engineering (see Sect. 7.4), application testing focuses on the produced artifacts of a product, and reuses where possible test artifacts from domain engineering, such as test reports, test cases, test case generators, syntax checkers, etc.

First of all, application testing checks whether the generated artifacts are syntactically correct, e.g., by reusing appropriate specifications or programs (e.g., syntax checkers) from domain testing. In cases where artifacts have a binary, application-specific form (e.g., Powerpoint files), the syntactic correctness of these artifacts has to be checked using the respective application (e.g., checking if the generated files can be opened with Powerpoint). Additionally, it must be made sure that the generation has not introduced unwanted errors (e.g., characters that were not there in the respective component in domain engineering) into the content, and that other required properties are not lost (e.g., animations in Powerpoint still function as in the components defined in domain engineering).

Thereafter, it has to be checked whether the specific content configuration of the created product is as demanded in application analysis. In addition, the information product has to be validated by asking the questions: is this what was really desired? Is the way the content is distributed in packages really desired to be this way? If problems arise, then iterations to earlier phases of application engineering should be considered, or appropriate feedback should be given to domain engineering.

8.5 Summary and Discussion

Application engineering takes advantage of the preparations made in domain engineering and thus simplifies and accelerates the creation of a digital information product; this is the payback of the investment in domain engineering. In addition, an information product is created in a systematic way and, due to the predefined specifications and constraints, synergy effects can be realized on a content reuse level through common features and feature configurations planned in advance.

The generative approach used in PLANT application engineering differs from other approaches (e.g., mass customization [HSI05]) in the sense that it is not possible to generate every conceivable information product. Restrictions for possible configurations come from the conceptual product line model, and features can be realized only with available versions from the respective core asset version graphs. These decisions are related to the philosophy of product lines that intentionally offer only a certain degree of individualization, and which restrict the overall variability in order to make the reuse of core assets more efficient, simplify the construction and maintenance, and thus increase the quality of generated artifacts.

Part III

Tool Support & Case Study

9

The Desktop Workflow Engine

This Chapter presents the Desktop Workflow Engine (DWE), a tool used to design and execute the construction workflow model in order to create digital information products. In particular, the purpose and the user interface are described, along with concepts which facilitate variant creation. Furthermore, the typical usage scenario and the internal tool architecture are outlined. The novel concepts are summarized and discussed at the end of the Chapter.

9.1 Purpose

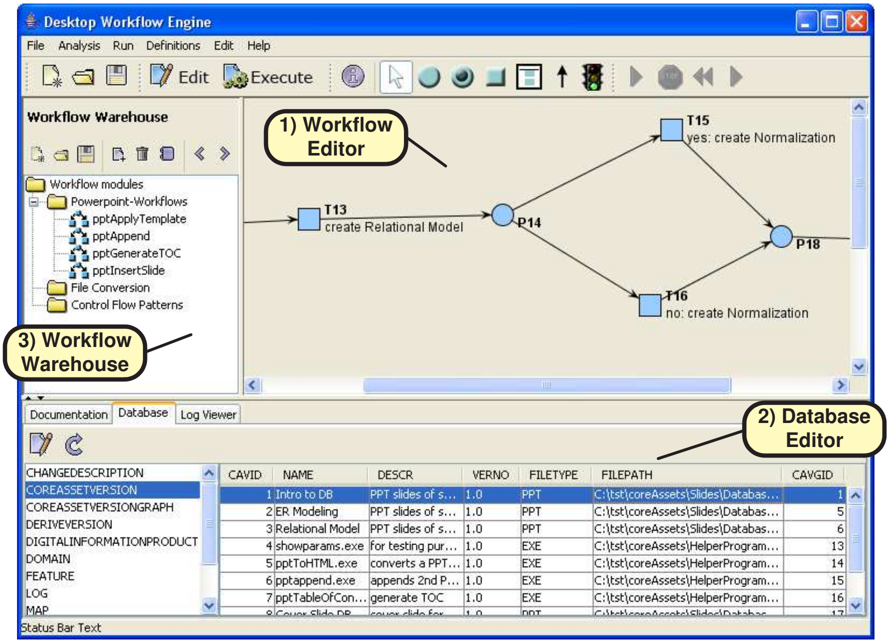
The DWE focuses on the systematic support of the technical construction process (or build process) of information products. It is used throughout the physical design phase in domain engineering as well as in application engineering. The DWE is a tool implemented in Java, and is intended to run on the personal desktop of a developer, e.g., of a domain engineer or application engineer.

In domain engineering, the DWE is used to create a workflow model, i.e., a QX net (Sect. 7.2.3), which represents the construction specification of the product line. The tool implements analysis algorithms to support the developer with the verification of the workflow model to make sure that only products with configurations allowed by the conceptual product line model can be constructed.

In application engineering, the DWE is used to make application-specific customizations and execute the workflow in order to construct a concrete digital information product. The DWE contains a seamlessly integrated relational database management system (DBMS), to support QX nets (recall that a QX net consists of an extension of workflow nets, and an associated relational database; see Def. 7.5). The usage of the DWE from the user interface point of view is presented next.

9.2 User Interface

The user interface as seen by domain and application engineers is shown in Fig. 9.1. At the top, there is a toolbar with buttons needed for workflow editing and execution. Furthermore, the user interface is divided into three areas: 1) the area of the workflow editor; 2) an area at the bottom in which the user can see the content of different tabs, such as documentation, database editor, or log viewer; 3) an area on the left showing the contents of a workflow warehouse. The usage of each of these areas is described in turn.



CHANGEDESCRIPTION
COREASSETVERSION
COREASSETVERSIONGRAPH
DERIVEVERSION
DIGITALINFORMATIONPRODUCT
DOMAIN
FEATURE
LOG
MAP

CAVID	NAME	DESCR	VERNO	FILETYPE	FILEPATH	CAVGID
1	Intro to DB	PPT slides of s...	1.0	PPT	C:\tst\coreAssets\Slides\Databas...	1
2	ER Modeling	PPT slides of s...	1.0	PPT	C:\tst\coreAssets\Slides\Databas...	5
3	Relational Model	PPT slides of s...	1.0	PPT	C:\tst\coreAssets\Slides\Databas...	6
4	showparams.exe	for testing pur...	1.0	EXE	C:\tst\coreAssets\HelperProgram...	13
5	pptToHTML.exe	converts a PPT...	1.0	EXE	C:\tst\coreAssets\HelperProgram...	14
6	pptappend.exe	appends 2nd P...	1.0	EXE	C:\tst\coreAssets\HelperProgram...	15
7	pptTableOfCon...	generate TOC	1.0	EXE	C:\tst\coreAssets\HelperProgram...	16
8	Cover Slide DB	cover slide for	1.0	PPT	C:\tst\coreAssets\Slides\Databas...	17

Fig. 9.1. Screenshot of the DWE as seen by domain and application engineers.

9.2.1 The Workflow Editor

The workflow editor can be in one of two modes: editing mode or execution mode. The editing mode is used in domain design to model the workflow. The execution mode is used for testing purposes in domain design, as well as in application realization to enact the workflow and construct an information product.

Editing Mode

In this mode, the workflow editor is used to draw or delete the places, transitions, arcs, and tokens of a QX net interactively with the mouse. It supports zoom-in/zoom-out and grouping of subnets; for the combination of existing workflow models with the currently edited workflow, the DWE uses join operators that are similar to those introduced in [PS05a].

The editor automatically assigns a ID number to each place or transition, which is unique in the currently edited model. Labels with a textual description can be attached by the user to places or transitions. For each transition, additional properties can be specified (as demanded in Def. 7.5): pre-SQL and post-SQL queries, a path to a program that is executed when the respective transition occurs, and parameters for the program, which can also be queried from the database. More details on properties are discussed in Sect. 9.3.1.

The designer is responsible that the workflow model can only construct information products with feature configurations allowed by the conceptual product line model. The DWE supports the designer in various ways to achieve this goal. The designer can reuse workflow patterns (e.g., as in Fig. 7.6) or predefined workflow modules, which are both stored in the workflow warehouse (Sect. 9.2.3). In addition, the DWE offers the possibility to create a marking graph based on the edited workflow[1]. There are additional in-editing-checks that make sure that the workflow model has also the properties of a workflow net (see Def. 7.4); for example, the user is presented in the toolbar traffic lights that signalize either in red or green if the respective properties are satisfied, and the execution mode is enabled or disabled accordingly.

Execution Mode

When the DWE is in execution mode, it supports the enactment of the QX net according to the occurrence rule for QX nets specified in Sect. 7.2.3. Based on the occurrence rule, an internal scheduler determines all transitions that are enabled under the current marking, and execution is done in the following way:

- if no transition is enabled, the execution stops;
- if exactly one transition is enabled, the transition is scheduled to occur (i.e., the associated SQL queries and the program are executed);
- if a conflict situation arises in which more than one transition is enabled at the same time, then the user is asked to chose exactly one of the enabled transitions to occur (see Fig. 9.2). In this situation, the user decides how the control flow should go on.

[1] Remark: there is an upper bound of displayable vertices depending on the available memory.

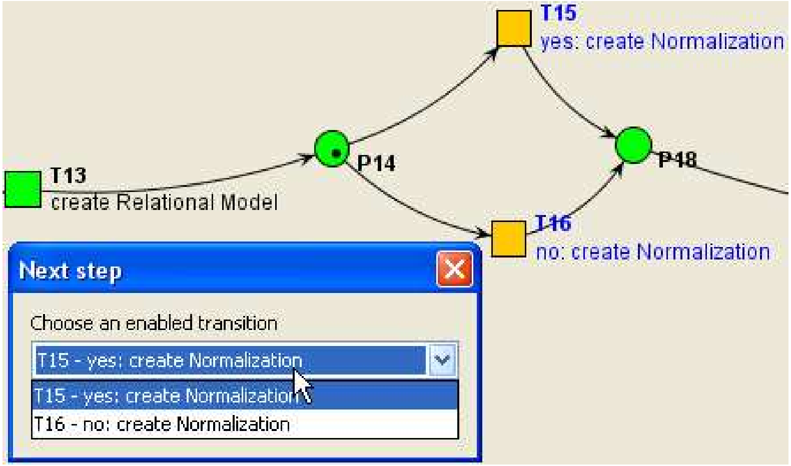
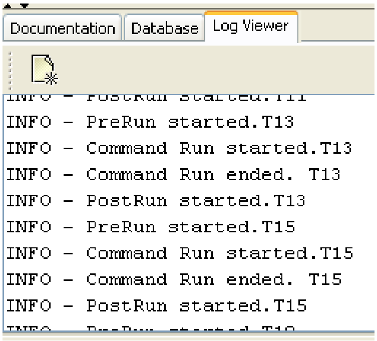

T15
yes: create Normalization
T13
create Relational Model
P14
P48
T16
no: create Normalization
Next step
Choose an enabled transition
T15 - yes: create Normalization
T15 - yes: create Normalization
T16 - no: create Normalization

Fig. 9.2. When more than one transition is enabled under a given marking in execution mode, the user is asked to chose one of the enabled transitions to occur.

When the scheduling is done in the aforementioned way, there are no transitions that are executed concurrently, and the execution of such transitions is “serialized”. Throughout the execution, the DWE automatically creates a system log which shows the order of occurrence of transitions (see Fig. 9.3).



```
INFO - PreRun started.T13
INFO - Command Run started.T13
INFO - Command Run ended. T13
INFO - PostRun started.T13
INFO - PreRun started.T15
INFO - Command Run started.T15
INFO - Command Run ended. T15
INFO - PostRun started.T15
```

Fig. 9.3. The DWE automatically records a log of occurring transitions.

The DWE stops the execution of a workflow in case of exceptions, e.g., if a program that is to be executed throws exceptions, or if SQL queries cannot be executed because of constraint violations in the database. The system log can be used in such cases to trace back the origin of the exceptions.

9.2.2 The Database Editor

The database editor provides a graphical interface to the database management system (DBMS) that is seamlessly integrated into the DWE. The DBMS is based on HSQLDB [ST05], which is a full-fledged relational DBMS implemented in Java, and its functionality is available within the DWE. The reader is referred to the handbook of HSQLDB [ST05] for details on features of the HSQLDB.

A specific database with its relations and integrity constraints is always associated to a specific QX net whose control flow is shown in the workflow editor, and the database is opened and saved together with the QX net[2]. Within the database editor, a user can execute SQL DDL commands [ST05] to create a database schema with appropriate constraints, or insert data tuples into the relational database. All names of existing relations in the database are shown within the database tab on the left side. A relation name can be selected with the mouse, and the content of the respective relation will be displayed with the attributes and values in a tabular form (Fig. 9.1).

For testing purposes, a user can also execute SQL selection commands [ST05] and display the results. Modification of tuples can be done via SQL update statements [ST05], or via the graphical user interface where the attribute values of a tuple can be modified by clicking on them and entering a new value. Such an easy update of data is important for application engineering, as pre-configured values defined in domain engineering remain easily adaptable in application engineering.

For example, a file path specification defined as an attribute value in a tuple can be adapted in the aforementioned way, and if the path value is queried inside one or more transitions during execution (e.g., to be passed as a parameter to some program), the updated value will be received. The advantage of separating such data from the internals of the control flow is that a domain engineer can specify the control flow in the workflow editor and initial parameter data in the database, while an application engineer is able to modify the data more easily in application engineering, without having to look at all internals of the construction process.

9.2.3 The Workflow Warehouse

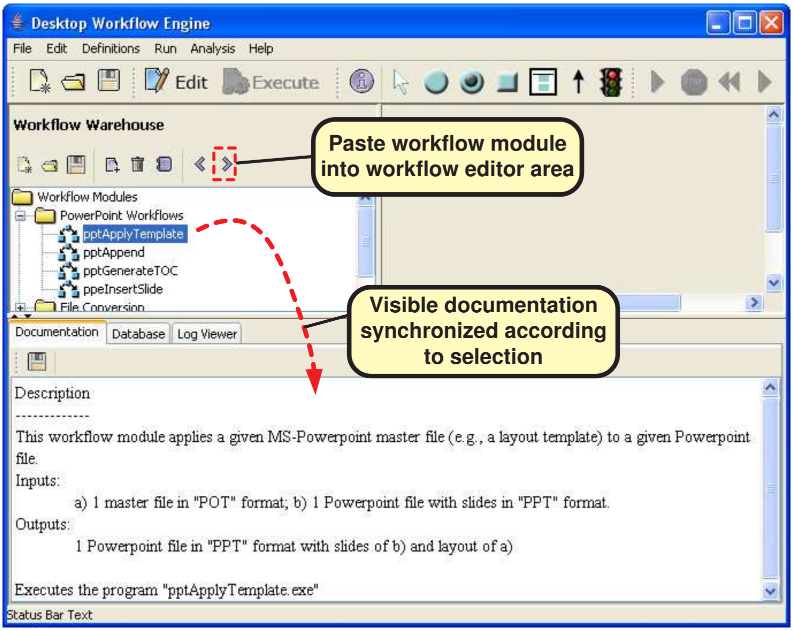
The workflow warehouse acts as a repository for predefined workflow modules or workflow patterns, and is independent of the currently edited QX net or database. The workflow warehouse can be saved and exchanged separately, and thus supports the reuse of frequently needed process parts (the usage scenario will be discussed in Sect. 9.4).

Inside the workflow warehouse, the user can define a hierarchical structure of folders which can contain workflow modules or other folders. A workflow

[2] The DWE even saves the workflow net and the database in the same file.

module in the warehouse is created by marking in the workflow editor a subnet and duplicating it into the warehouse; the DWE automatically stores also the properties of the respective places and transitions in the warehouse. It is also possible to copy places and transitions with empty properties, so that the created workflow module represents a workflow pattern that can be extended later. To reuse a workflow module from the warehouse, it has to be selected by the user and copied to the editor area. The DWE inserts the module into the edited workflow; at first, it is placed on an empty area and stays disconnected, and the DWE reassigns its IDs of places and transitions so that they do not collide with existing ones. Thereafter, the user has to make the desired connections between the existing workflow and the inserted module.

The easy creation and access of the documentation of workflow modules is vital for their reuse. The DWE supports this with the documentation tab at the bottom area, which shows the documentation of the currently highlighted folder or workflow module in the workflow warehouse (Fig. 9.4). It is also possible to edit and modify the documentation right away; it is finally stored together with the workflow warehouse.



Description

This workflow module applies a given MS-Powerpoint master file (e.g., a layout template) to a given Powerpoint file.
Inputs:
a) 1 master file in "POT" format; b) 1 Powerpoint file with slides in "PPT" format.
Outputs:
1 Powerpoint file in "PPT" format with slides of b) and layout of a)

Executes the program "pptApplyTemplate.exe"

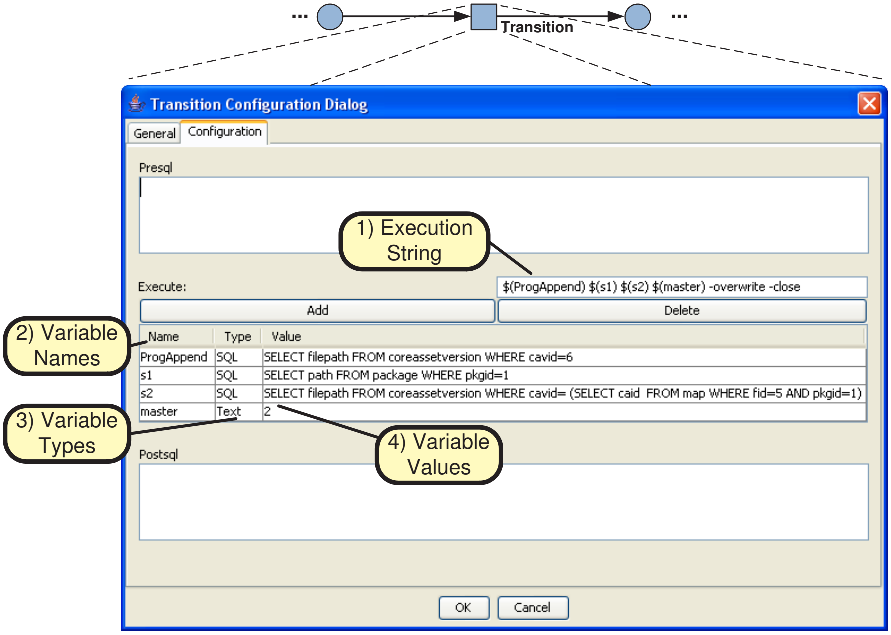
Fig. 9.4. The integrated documentation of the workflow warehouse.

9.3 Concepts Supporting Parametrization and the Creation of Variants

The DWE supports the creation of variants on an implementation level with various concepts that are discussed next.

9.3.1 Transition Properties

It has already been mentioned that transitions have internal properties. An example configuration dialog for one transition with its internal properties is depicted in Fig. 9.5.



Execute: $(ProgAppend) $(s1) $(s2) $(master) -overwrite -close

Name	Type	Value
ProgAppend	SQL	SELECT filepath FROM coreassetversion WHERE cavid=6
s1	SQL	SELECT path FROM package WHERE pkgid=1
s2	SQL	SELECT filepath FROM coreassetversion WHERE cavid= (SELECT caid FROM map WHERE fid=5 AND pkgid=1)
master	Text	2

Fig. 9.5. Properties of a transition.

The "Execute" input line can store a string with the path of an executable program and its related execution parameters; the program is executed when the transition occurs. For example, "*c:\pptappend.exe file1.ppt file2.ppt 2 -overwrite -close*" would call a program that first converts the layout of the slides of *file1* to the layout of *file2*, appends thereafter all slides of *file2* after the last slide of *file1*, and finally saves the modifications made in *file1*. This "hard coded" definition works, however, it is difficult to update when for example different file names have to be used for different product variants.

Local and Global Variables

To simplify variant creation, local variables can be defined within a transition, using the transition configuration dialog. Each variable has a name, a type, and a value. Name and value must be a string, and possible types are "text" or "SQL". The variable values are not allowed to contain references to other variables. At design time, placeholders for variables (e.g., "$(s1)" for variable "s1") are inserted by the developer in the execution string. At run-time, when the transition occurs, the DWE replaces the placeholders by the string values bound to the respective variables[3]. The variables are evaluated in the order of definition, and the placeholders are replaced with the respective value also in this order. This replacement is done before the execution string is executed.

For a variable of type "text", its name is bound to a fixed string value, e.g., the variable "master" in Fig. 9.5 is permanently bound to the string "2". Such variables (which are actually constants) may help to convey the meaning of certain parameter values within the execution string.

For a variable of type "SQL", a valid SQL query is specified at design-time in its value field, whose result after execution in the database must be a string. At run-time, the name of the SQL-type variable is bound to the string which results from the execution of the query. For example, the query associated to the variable "ProgAppend" in Fig. 9.5 could evaluate at run-time to the string "c:\coreassets\HelperPrograms\pptAppend.exe", which is obtained from the respective relation in the database.

A similar dialog allows the definition of global variables that are accessible from all transitions in the workflow. Similar to local variables, global variables have a name, type, and value. The type can be either "text" or "SQL", and a value is not allowed to contain references to other variables. The global variables are evaluated when the workflow execution is initiated. During the replacement of placeholders in the execution string of a transition, its local properties environment with local variables is searched first, i.e., if there is a local and global variable with the same name, the value of the local one will be taken; if a variable name is not found in the local properties but declared as a global variable, the respective global value is taken. Should the execution string of a transition contain a variable name that is neither declared locally nor globally, an error message is shown and the workflow execution is stopped.

The concept of variables, especially that of SQL-type variables that obtain values from the database, simplifies the variant creation for information products. Especially parameter data that is passed on to helper programs can now be stored and modified separately in the database. This data can be modified independently of the control flow structure, without having to look at the internal details of transitions. In addition, an updated attribute value is automatically used in all locations querying the respective data. Finally, the database has at the same time a documentation function.

[3] This corresponds to a call by value (see [Lou02, ALSU07]).

Pre-SQL and Post-SQL Statements

Pre-SQL and post-SQL statements are SQL update statements [ST05] which can be defined within one transition (see also 7.2.3). If specified, a pre-SQL statement is executed before the execution of the respective execution string, and a post-SQL statement after it.

When a particular information product is created in execution mode, different sets of transitions may be executed, depending on the choices made by the application engineer. Pre-SQL and post-SQL statements can be used to maintain a more detailed execution log, and to document in an automated way the features selected for a variant. For example, a pre-SQL statement could insert in the log a tuple with details on the selected features, and the post-SQL statement logs the successful creation. Using a delete statement in the pre-SQL of a transition at the start of the workflow, the execution log can be automatically cleared in case that the workflow is re-executed.

9.3.2 Helper Programs

The adaptivity of the DWE to different application areas is achieved by separating the workflow engine from the programs that actually create a variant of an information product. These helper programs are called by the workflow engine (similar to [Hol95]), and their behavior is controlled by the parameters passed on to them.

The helper programs are used for example to compose a variant out of construction artifacts, to convert or copy files, add metadata, etc. Helper programs can also be wrappers constructed around other applications that should be integrated into the automated workflow, without the user being aware of them. Conceptually, such wrappers are an additional software layer between the DWE and an external application. Technically, wrappers are themselves executable programs that expose a predefined interface to the DWE. When executed by the DWE, a wrapper internally calls the external application, possibly passing data to it and processing results (see Fig. 9.6). The wrapper also catches exit codes from the application after its execution and reports the success or possible exceptions to the DWE.

An example of such a wrapper is the "pptAppend.exe" program which is used to integrate the MS-Powerpoint application into an automated workflow. This program can be called with two PPT files as parameters, and the slides of the second file are appended to the first file. Internally, this is accomplished by calling the installed Powerpoint application; this automation is based on the Component Object Model (COM) [Tur00].

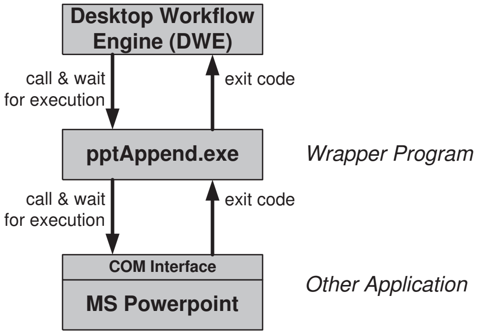


Fig. 9.6. Integrating external applications with wrappers into the DWE.

9.4 Usage Scenario

The typical usage scenario of the DWE is depicted in Fig. 9.7 from a logical point of view which also clarifies how, with which data, and by whom the DWE is used.

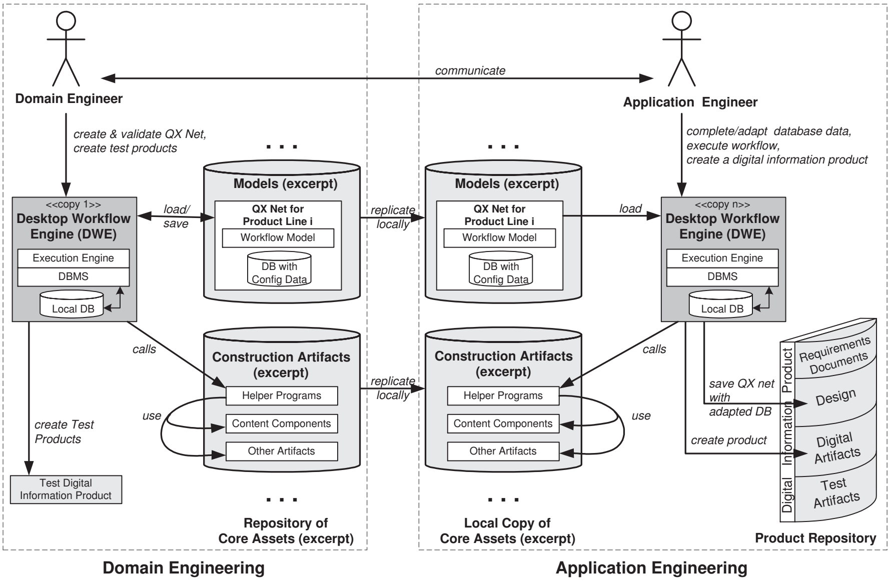


Fig. 9.7. Scenario for the usage of the DWE.

Every developer (i.e., domain or application engineer) is intended to have a copy of the DWE that runs on his/her personal desktop. The data that is needed by the DWE has to be available locally, and the DWE produces only local data that can be thereafter copied by the developer to other locations.

A domain engineer creates a QX net specifying the construction workflow model for all products in one product line. The result output by the DWE is a file that contains both the workflow model and the associated database with configuration data. Sample information products are created in domain testing in order to assess the construction workflow model. The artifacts that are needed during the execution of the QX net, e.g., helper programs or other artifacts, come from the construction artifacts section of the core assets repository (see Figs. 5.1 and 9.7). The final file of the QX net that is eventually released for application engineering is stored in the models section of the repository of core assets (cf. Figs. 5.1 and 9.7).

In application engineering, an application engineer replicates the core asset base of domain engineering locally. Thereafter, he or she loads the QX net into his local DWE and adapts the data in the database (e.g., the product map; Fig. 7.5) to suit the individual product. The adapted QX net and the artifacts of the product resulting from the execution of the QX net are stored in an individual product repository. Finally, the application engineer reports the product configuration to the person responsible for family engineering[4].

In some scenarios it is possible that only one single person does both domain and application engineering, so that replication or coordination is not problematic. However, in collaborative scenarios several several people will access the core asset repository, and each application engineer who creates a product variant will have his own locally replicated application engineering environment (depicted in the right part of Fig. 9.7). In such situations, concurrency control or consistency problems arise. These aspects are not discussed here in detail, since various tools and protocols already exist to handle such problems (e.g., see [Fei91, CSFP04]).

9.4.1 Differences from Traditional Workflow Management Approaches

Traditional workflow modeling approaches focus on the automation of business processes in collaborative scenarios with well-structured processes[5] [LR99, AH02a]. In such contexts, the workflow model is assumed to be stable, and the automation is based on exactly one centralized workflow model. In addition, such workflow models are often extended by role models which specify who should perform certain tasks. The typical architecture is designed with

[4] Note: this connection is omitted in Figs. 9.7 and 5.1 to avoid an overload of connections.

[5] By contrast, if collaboration is not structured or predictable at all, groupware systems are frequently employed, which do not control a process execution and which typically focus more on various forms of group communication (cf. [LR99, AH02a]).

a centralized workflow engine which has total control to enact the model. Attached clients are controlled by the engine that assigns them pieces of work; these are to be processed by persons using the clients.

This traditional approach is considered to be too “heavy weighted” and inflexible for the PLANT context for several reasons. PLANT does not attempt to model the complete development process, but concentrates only on the well-structured build process of information products, and thus has a smaller and more specialized focus. In this context, a centralized workflow model with centralized enactment is unsuitable. Other approaches with a decentralized enactment (e.g., [VW99]) do not fit well enough into the specific context of product lines, as configurations of variants were not in the original focus.

Product lines delay product-specific decisions up to the latest possible point in application engineering. Despite the fixed control flow structure of the workflow model, the product map (which is used within the workflow) needs to be configured individually for every product, as already shown, in order to create an information product out of the specifically desired versions of core assets. Thus, it would be inefficient to model all possibilities in a centralized fashion right away, since until application engineering, it is not known which versions of core assets will be used to implement which product features.

PLANT therefore follows a novel approach and splits up the workflow model development. In domain engineering, the workflow model is prepared for later execution. Then, in application engineering, adaptations are made with respect to the data that is to be used, and the model is enacted to construct products. Because of its specialized focus, the workflow model does not use role models. The control is decentralized, as every developer can use a DWE (Fig. 9.7) – this differs also from traditional approaches that have only one centralized workflow engine. Although product-specific adaptations of the QX net are made locally, this does not mean that inconsistencies are produced, because application engineers are only allowed to add or change certain data in the database (see Fig. 7.5). In addition, predefined integrity constraints in the database prohibit invalid changes. Due to the specialized focus of the construction workflow model in PLANT, fewer participating persons can be expected, compared to traditional workflow management scenarios.

9.5 Outline of the Internal Architecture

The internal architecture of the DWE is briefly outlined using a UML package diagram (Fig. 9.8). The packages contain Java classes and/or other packages, and the dashed arrows show dependencies between the packages based on method calls or field access (obtained using [Cre06]).

Internally, it is distinguished between the model part that deals with conceptual data, the view part that displays model data, and the control part

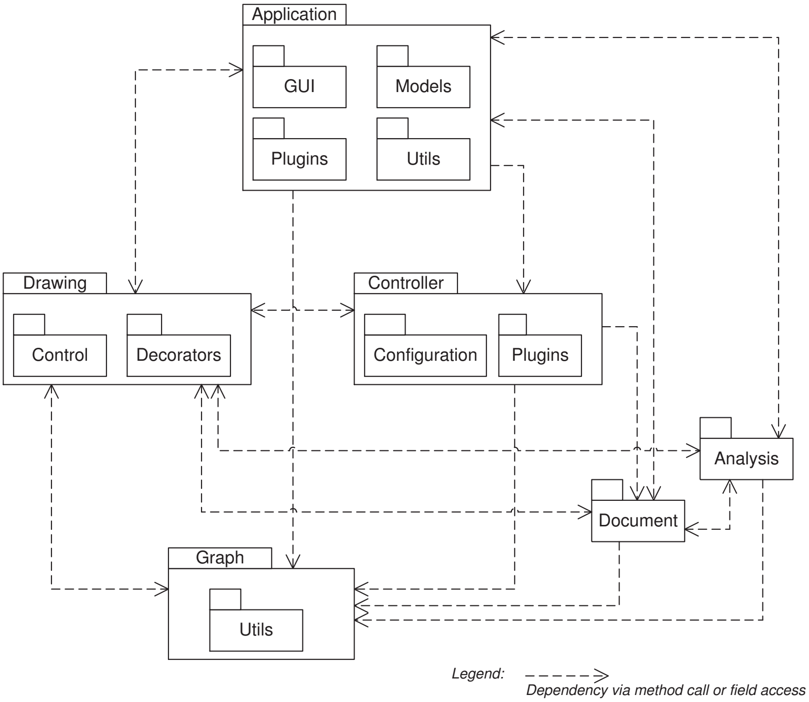


Fig. 9.8. Package diagram outlining the architecture of the DWE.

that manipulates model data and which sends it to display. An excerpt of the functionality that is implemented by each package is described next:

Application package. Contains: the main class of the application, the Database Management System (DBMS) based on HSQLDB [ST05], and various data structures from the Jakarta Commons library [Jak06]. The subpackage *GUI* contains the classes for the Graphical User Interface (GUI) based on Java SWING [EEL^{+}02], such as editor frame, workflow warehouse tree, SQL editor, configuration panels. The subpackage *Models* contains the application model with data that does not belong to the document model. The subpackage *Utils* has classes implementing copy & paste functionality for workflow modules from the warehouse to the workflow editor and vice-versa. The subpackage *Plugins* contains various elements used in the main application windows, e.g., the menu bar, or the tool bar.

Document package. Contains: classes with data of the workflow graph, its layout and tokens, the associated relational database, and storage routines.

Graph package. Contains: the data related to a graph, such as classes for edge, vertex, place, transition, tokenlist; it extends classes from the JUNG framework [JUN06]. The contained package *Utils* provides functionality related to the logical structure of the graph, e.g., the check that a workflow model satisfies the conditions of a workflow net.

Drawing package: Contains: classes for visualization of model data, the drawing panel of the workflow editor, picking and grouping of graph elements; it is based on the JUNG framework. The subpackage *Control* encapsulates the logic for user interaction, e.g., reacting to mouse events. The subpackage *Decorators* influences the appearance of nodes in the graph, for example depending whether they are selected in the editor area or not.

Controller package: Contains: classes implementing different schedulers for workflow execution, e.g., for execution step-by-step (for debugging purposes) or automatic execution (for regular execution) until more than one transition is enabled. The subpackage *Configuration* provides helper routines needed during execution, e.g., which transitions are enabled under a given marking. The subpackage *Plugins* provides, based on an extensible architecture, the logic defining what should happen during execution, and how to handle errors. Finally, the controller package contains classes maintaining the state of the workflow, the symbol table for variables, the execution logic for SQL statements, and logging capabilities (supported by the Log4J framework [LOG06]).

Analysis package: Contains: classes that implement analysis algorithms on the workflow model, e.g., construction and display of a marking graph, reachability analysis with linear-algebraic techniques (based on [Mur89, DR98]). This functionality can be used by domain engineers during the creation of a workflow model.

9.6 Summary and Discussion

This Chapter illustrates the Desktop Workflow Engine which is a tool that is used to create the construction workflow model for information products in domain engineering, and to enact this model in order to create individual products in application engineering. The DWE focuses entirely on the technical build-process of products and integrates various concepts that facilitate the variant creation on an implementation level. In addition, the DWE simplifies process reuse, as the files of a workflow warehouse or a QX net can be simply exchanged, for example even by e-mail or through Web repositories. The adaptation of such exchanged processes to the local context of a developer is eased through the concepts of variables, as a developer only has to modify the appropriate values (e.g., for program path specifications), and not the structure of the workflow model.

The DWE is designed to be extensible for future additions. For example, additional schedulers can be added that handle the execution of the workflow model in a different way, which can execute it without user interaction or with concurrency (e.g., by defining for a certain model an execution plan in advance). Additional types of variables such as system-defined variables can also be introduced, which are bound for example to the current time, date, or version of used operating system; such variables can be used to pass on system context values to the called programs. The analysis capabilities are extensible as well, for example with respect to additions when concurrent execution is to be allowed.

The concepts of the Desktop Workflow Engine are general enough to be applied also in other contexts. For example, in a single source publishing context [TR05] the DWE can be used to implement the logic of content composition, transformation, and publication. The DWE can also be used in conjunction with the Personal Software Process [Hum02]. For software product lines, the DWE could replace other build-tools (e.g., [The07]) and simplify the creation of software products, as the called helper programs could be code generators or compilers. When conditional compilation is used to create different software products (see also Sect. 4.3.3), the appropriate parameters can be obtained from the database associated to a QX net. The analysis capabilities of the DWE also fill an existing gap in the software product line research, where the analysis of the technical build-process of products is often neglected.

10

Case Study

This Chapter presents a case study which shows how the PLANT approach has been applied in a real-life context. It focuses on the creation of one product line for digital information products in an educational context at the Business Information and Communication Systems Group in the AIFB Institute at the University of Karlsruhe, Germany. The purpose of this case study is to describe a specific example in a detailed way, to demonstrate the applicability of PLANT, and to validate the proposed approach. Finally, the results of the application of PLANT are evaluated.

10.1 Preparation

This case study follows the guidelines proposed in [Yin03, ZMM06]. The main research question that is to be answered is how exactly PLANT is applied to create a product line for digital information products in a "real world" situation. Although the case study is mostly descriptive, it has also explanatory aspects, as it explains at certain points why some specific decisions are made, as well as the rationale behind the respective choice.

The unit of analysis is one product line of digital information products in the specific educational context that is presented in the sequel. It is considered that this context reflects in most parts the typical problems encountered when a product line of digital information products is created.

Validity is constructed by collecting evidence from the following sources: application context, artifacts created with PLANT (e.g., models created by a domain engineer, artifacts created with the DWE tool presented in Chapter 9), available documents where PLANT has been applied and where it has not been applied, observation of improvements (in qualitative or quantitative terms).

To evaluate the PLANT Approach, the observation of improvements (in qualitative or quantitative terms) are compared to those in the situation without PLANT.

10.2 Usage Context and Encountered Problems

This case study is conducted within the Business Information and Communication Systems Group (BIC) at the AIFB Institute of the University of Karlsruhe, Germany. The BIC group offers several lectures related to the area of Information Systems. The ones of interest in this study are: Applied Informatics I, Workflow Management, and Database Systems. In addition to the lectures in Karlsruhe, the group offers a Database Systems course with a different topic emphasis at the Vienna University of Economics and Business Administration.

The educational material for these lectures is created in MS Powerpoint format. A Powerpoint file typically contains the content for a certain topic, and constitutes a learning object. The Powerpoint content format is used by lecturers and assistants, and the files are often converted to the PDF format and stored on a Web server for delivery to students. Metadata is captured with the metadata attributes of the Powerpoint or PDF format. The students have the opportunity to do blended learning, i.e., they can either do e-learning at home by using only the PDF files, attend the lecture with the slides printed out, or both.

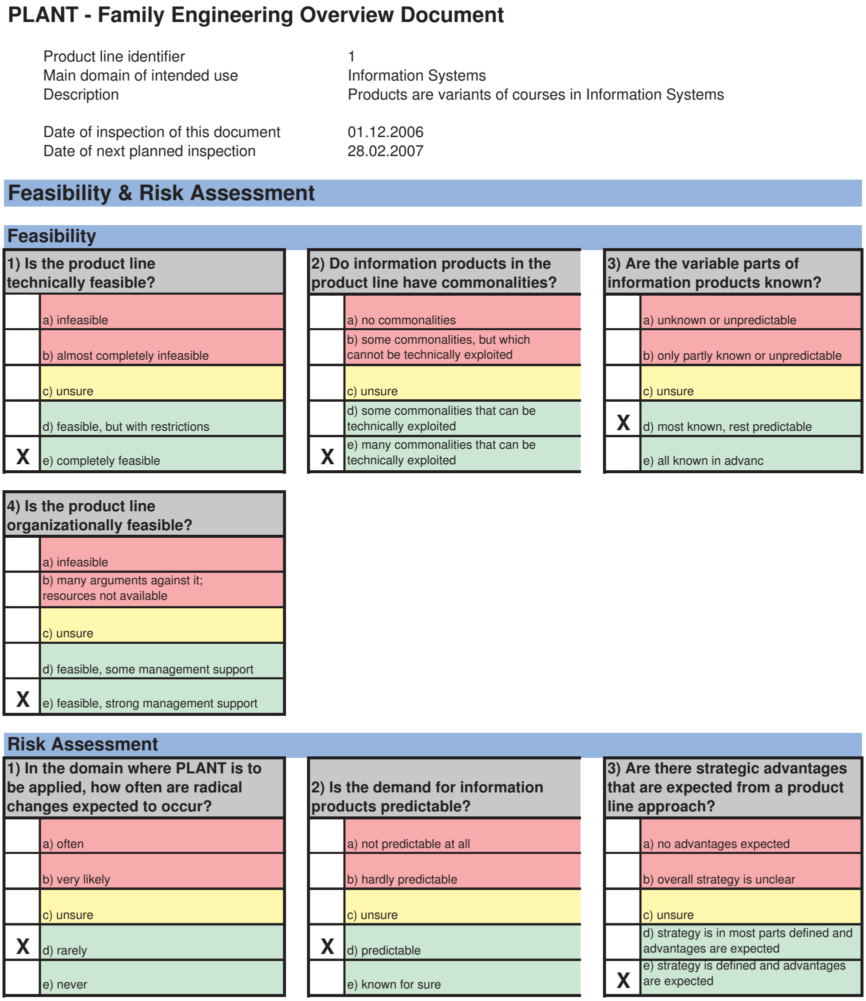
The aforementioned lectures have commonalities in some parts, which cause problems when the content has to be updated by different assistants, as it is often the case that the updates of common parts are not propagated to the other lectures systematically enough. For example, typical updates which occur rather often are: corrections, insertion of suggestions, improvements of parts that were difficult to understand, or additions with new developments in the fields. Another problem is that the content of the common parts may be reworked in a different way by different assistants, so that the appearance of the common parts may diverge as well during the evolution of the content. Finally, the creation and update of educational material with commonalities is inefficient, as these commonalities are not used somehow to achieve synergy effects.

10.3 Family Engineering

This Section illustrates how family engineering (see Chapter 6) was performed for this special case. The created family engineering overview document is presented and explanations are given for the choices.

Feasibility and Risk Assessment

Figure 10.1 shows the initial part of the family engineering overview document, which focused on feasibility assessment and risk assessment. The rationale behind the choices in the context of the case study is explained below.



PLANT - Family Engineering Overview Document

Product line identifier	1
Main domain of intended use	Information Systems
Description	Products are variants of courses in Information Systems
Date of inspection of this document	01.12.2006
Date of next planned inspection	28.02.2007

Feasibility & Risk Assessment

Feasibility

	1) Is the product line technically feasible?
	a) infeasible
	b) almost completely infeasible
	c) unsure
	d) feasible, but with restrictions
X	e) completely feasible

	2) Do information products in the product line have commonalities?
	a) no commonalities
	b) some commonalities, but which cannot be technically exploited
	c) unsure
	d) some commonalities that can be technically exploited
X	e) many commonalities that can be technically exploited

	3) Are the variable parts of information products known?
	a) unknown or unpredictable
	b) only partly known or unpredictable
	c) unsure
X	d) most known, rest predictable
	e) all known in advanc

	4) Is the product line organizationally feasible?
	a) infeasible
	b) many arguments against it; resources not available
	c) unsure
	d) feasible, some management support
X	e) feasible, strong management support

Risk Assessment

	1) In the domain where PLANT is to be applied, how often are radical changes expected to occur?
	a) often
	b) very likely
	c) unsure
X	d) rarely
	e) never

	2) Is the demand for information products predictable?
	a) not predictable at all
	b) hardly predictable
	c) unsure
X	d) predictable
	e) known for sure

	3) Are there strategic advantages that are expected from a product line approach?
	a) no advantages expected
	b) overall strategy is unclear
	c) unsure
	d) strategy is in most parts defined and advantages are expected
X	e) strategy is defined and advantages are expected

Fig. 10.1. The "feasibility and risk assessment" part of the family engineering overview document.

Feasibility

- **1) Is the product line technically feasible?**
  Choice: *e*) completely feasible
  Explanation: the relevant file formats (Powerpoint PPT, PDF) could be handled; wrapper programs for Powerpoint existed which could append files or apply a given layout template automatically; conversion of PPT to PDF was possible; PPT files of current courses were available; the necessary technical infrastructure existed

- **2) Do information products in the product line have commonalities?**
  Choice: *e*) many commonalities that can be technically exploited
  Explanation: after inspecting the available material, the existing courses were found to have in several parts slides that occur identically or almost identically in different courses; these slides could be extracted and modularized; therefore, there was potential for synergy effects.

- **3) Are the variable parts of information products known?**
  Choice: *d*) most known, rest predictable;
  Explanation: there existed regularities between the common and differing parts in courses, and they could also be described on a conceptual level; the parts of the required variants could be deduced from the conception of the courses

- **4) Is the product line organizationally feasible?**
  Choice: *e*) feasible, strong management support
  Explanation: the necessary staff resources were available, the product line strategy was accepted and supported.

Risk assessment

- **1) In the domain where PLANT is to be applied, how often are radical changes expected to occur?**
  d) rarely;
  Explanation: the domain was typically stable and expected to remain stable, as the courses were part of a curriculum that was in line with the overall department and faculty strategy. The employed file formats were widely used and were likely to be available in the future.

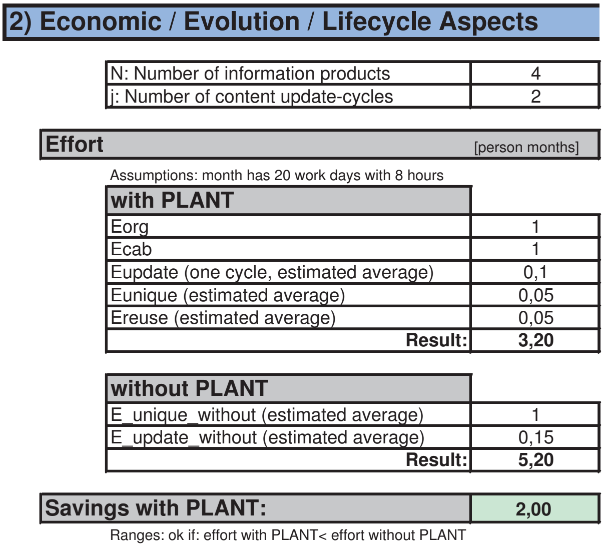
- **2) Is the demand for information products predictable?**
  d) predictable
  Explanation: the demand for certain course variants was predictable, as the courses were integrated in a curriculum and offered on a regular basis.

- **3) Are there strategic advantages that are expected from a product line approach?**
  e) strategy is defined and advantages are expected
  Explanation: main aspects of the strategy were to ease the creation of course variants, make the creation process easier and less error prone.

According to the criterion defined in Sect. 6.1, all answers were either d) or e), so that the feasibility and risk of this product line were within acceptable ranges.

Economic, Evolution, Lifecycle Aspects

Figure 10.2 illustrates the part of the family engineering overview document that focused on a quantitative effort estimation and comparison with/without PLANT. The total effort with/without PLANT was calculated according to the formulas discussed in Sect. 6.2.1 and 6.2.2[1].



2) Economic / Evolution / Lifecycle Aspects

N: Number of information products	4
j: Number of content update-cycles	2

Effort [person months]

Assumptions: month has 20 work days with 8 hours

with PLANT	
Eorg	1
Ecab	1
Eupdate (one cycle, estimated average)	0,1
Eunique (estimated average)	0,05
Ereuse (estimated average)	0,05
Result:	**3,20**

without PLANT	
E_unique_without (estimated average)	1
E_update_without (estimated average)	0,15
Result:	**5,20**

Savings with PLANT:	2,00

Ranges: ok if: effort with PLANT< effort without PLANT

Fig. 10.2. The part on economic, evolution, lifecycle aspects of the family engineering overview document.

The rationale behind the estimation was as follows:

- there were 4 courses mentioned in Sect. 10.2, which made up at the beginning 4 information products in the product line
- the number of update cycles was derived from the context, and it was assumed that there would be 2 content update cycles, e.g., after each semester
- $E_{org}, E_{cab}, E_{update}, E_{unique}, E_{reuse}$ were estimated by a domain expert (only one was available) after viewing existing material.

[1] Reminder:

$$E_{with} := E_{org} + E_{cab} + N * (E_{reuse_{with}} + E_{unique_{with}} + j * E_{update_{with}})$$

$$E_{without} := N * (E_{unique_{without}} + j * E_{update_{without}})$$

- the effort estimations without PLANT were based upon experience of the domain expert and his observations during the last three years before this case study.

It was estimated that in this case the savings with PLANT were about 2 person months. As an aside, the average effort to create one additional product with PLANT was: $E_{reuse_{with}} + E_{unique_{with}} + j * E_{update_{with}} = 0.3$ person months; the average effort to create one additional product without PLANT was: $E_{unique_{without}} + j * E_{update_{without}} = 1.3$ person months. This meant that even if the number of information products in the product line increased during the evolution, the effort with PLANT would be lower than that without PLANT.

Configuration Management

During the initiation of PLANT there were no configurations there yet for the market-based view and product-based view, and the evaluation of a match between the market-based and product-based view was overridden at that point.

After the first pass of application engineering (to be described in Sect. 10.5) it was evaluated that the realized product configuration indeed matched the market-based view of demanded configurations (to be described in Sect. 10.4), and "match" meant that all features from the market-based view were contained also in the product-based view.

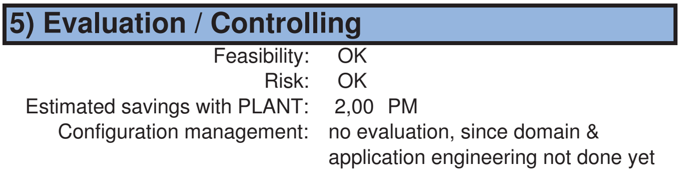
Organization

The product line in this case study had few products, and for the presented context the development department model was chosen as the organizational form (cf. Sect. 4.5.2) in which every staff member can do any type of work. The effort for the required tasks could be handled by one single person who was responsible for family engineering, domain engineering, and application engineering.

Evaluation and Controlling

Figure 10.3 shows the section of the family engineering overview document which summarized the previous results during the initiation of PLANT, to ease the decision making whether to create a product line or not. As feasibility and risk were within acceptable ranges and the usage of PLANT brought effort savings, the decision was in favor of PLANT.

The data was re-evaluated for controlling purposes after the first pass of domain engineering (to be described in Sect. 10.4) and the first pass of application engineering (to be described in Sect. 10.5). After domain engineering, there were no changes to the aforementioned data, and the product line proved



5) Evaluation / Controlling

Feasibility:	OK
Risk:	OK
Estimated savings with PLANT:	2,00 PM
Configuration management:	no evaluation, since domain & application engineering not done yet

Fig. 10.3. The "evaluation and controlling" part of the family engineering overview document (during initiation of PLANT).

to be feasible. After application engineering, configuration management data was available for the product-based view, and as already mentioned it was evaluated that configurations from product-based view and market-based view matched.

10.4 Domain Engineering

This section illustrates how domain engineering (see Chapter 7) has been applied.

Domain Analysis

- **Domain scoping**
  - *Product portfolio scoping:* the information products of the product line were variants of Information Systems courses that were adapted to the special context of the curriculum for which the department had to offer courses.
  - *Information domain scoping:* the information domains relevant for this special educational context are sketched in Fig. 10.4, and Information Systems was the root domain.

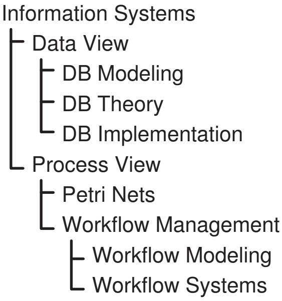
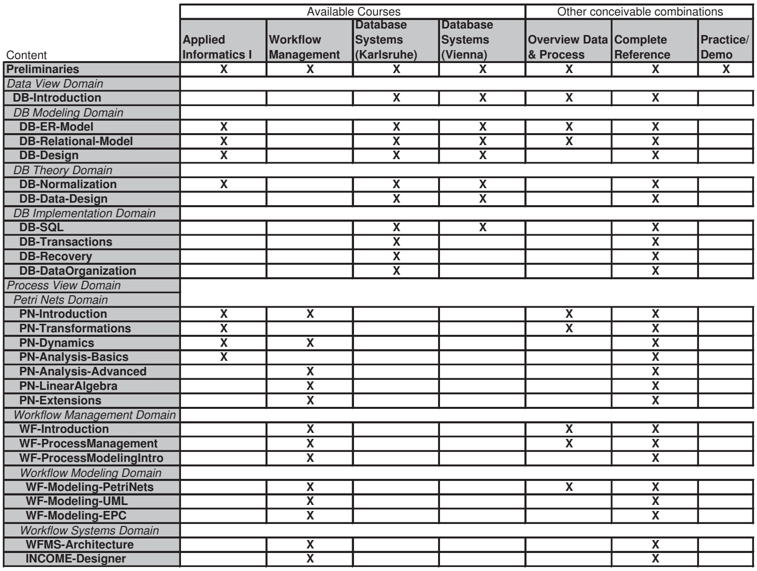


Fig. 10.4. Relevant domains obtained during information domain scoping.

– *Asset scoping:* the core assets that were content components were slides in Powerpoint format.

Next, the requirements for the product line were captured. For the content requirements, Fig. 10.5 depicts the topics which were treated in available courses, and groups them according to the domains obtained in information domain scoping. In addition, other conceivable combinations which were useful in the department were captured as well, e.g., an information product that contained all components (which could be used for reference purposes), or a product that contained only the preliminaries (used for example in lectures where some tool was demonstrated). More details on the requirements are described in the following list, which resulted from a first investigation, in natural language; this list was used later to create more abstract models:



	Available Courses				Other conceivable combinations		
Content	Applied Informatics I	Workflow Management	Database Systems (Karlsruhe)	Database Systems (Vienna)	Overview Data & Process	Complete Reference	Practice/ Demo
Preliminaries	X	X	X	X	X	X	X
Data View Domain							
DB-Introduction			X	X	X	X	
DB Modeling Domain							
DB-ER-Model	X		X	X	X	X	
DB-Relational-Model	X		X	X	X	X	
DB-Design	X		X	X		X	
DB Theory Domain							
DB-Normalization	X		X	X		X	
DB-Data-Design			X	X		X	
DB Implementation Domain							
DB-SQL			X	X		X	
DB-Transactions			X			X	
DB-Recovery			X			X	
DB-DataOrganization			X			X	
Process View Domain							
Petri Nets Domain							
PN-Introduction	X	X			X	X	
PN-Transformations	X				X	X	
PN-Dynamics	X	X				X	
PN-Analysis-Basics	X					X	
PN-Analysis-Advanced		X				X	
PN-LinearAlgebra		X				X	
PN-Extensions		X				X	
Workflow Management Domain							
WF-Introduction		X			X	X	
WF-ProcessManagement		X			X	X	
WF-ProcessModelingIntro		X				X	
Workflow Modeling Domain							
WF-Modeling-PetriNets		X			X	X	
WF-Modeling-UML		X				X	
WF-Modeling-EPC		X				X	
Workflow Systems Domain							
WFMS-Architecture		X				X	
INCOME-Designer		X				X	

Fig. 10.5. Some of the required content configurations as seen from the demand-side perspective.

The detailed requirements are shown next. They were used as a basis to create more abstract models, such as the conceptual product line model or the construction specification (cf. Figs. 10.6 and 10.8).

- **Content requirements**
  – **Common requirements:**
  all courses had *Preliminaries* with organizational details; courses with *DB Modeling Domain* contained *DB-ER-Model* and *DB-Relational-Model*; courses with the *Petri Nets Domain* contained *PN-Introduction*

in common; courses with the *Workflow Management Domain* commonly covered *WF-Introduction* and *WF-ProcessManagement*; courses with *Workflow Modeling Domain* had *WF-Modeling-PetriNets* in common;

  - **Optional requirements:**
    in the *Data View Domain*, *DB-Introduction* was optional; in the *DB Modeling Domain*, *DB-Design* was optional; in the *DB Theory Domain*, *DB-Normalization* and *DB-Data-Design* were optional; in the *DB Implementation Domain*, *DB-SQL, DB-Transactions, DB-Recovery, DB-DataOrganization* were optional; in the *Petri Nets Domain*, *PN-Transformations, PN-Dynamics, PN-Analysis-Basics, PN-Analysis-Advanced, PN-LinearAlgebra, PN-Extensions* were optional; in the *Workflow Management Domain*, *WF-ProcessModelingIntro* was optional; in the *Workflow Modeling Domain*, *WF-Modeling-UML, WF-Modeling-EPC* were optional; in the *Workflow Systems Domain*, *WFMS-Architecture,*
    INCOME-Designer were optional.
  - **Constraints:**
    if *Data View Domain* was treated, then the *DB Modeling Domain* had to be treated, too; if the *Process View Domain* was treated, then the *Petri Nets Domain* had to be treated; if the *Workflow Management Domain* was treated, then the *Workflow Modeling Domain* had to be treated
  - **Content component model requirements:**
    a content component had to be realized as a Powerpoint file.
  - **Crosscutting concerns:**
    a specific course had to use a predefined layout template, which was available as a Powerpoint master file.

- **Packaging requirements**
  A package was a Powerpoint file with an empty layout, which contained only one title slide. The package file was used to append content components into it, using a predefined sequence. The sequence of content components within a package had to be preserved even if some optional components were not included. In case that a package remained empty (because the chosen content configuration did not select a component), the package was deleted. The list below lists the packages that had to be created, along with the specification which potential content components (from Fig. 10.5) they could contain, as well as the sequence of occurrence of the content components within the packages. An optional requirement was that a package could contain a table of contents at the beginning, which has to be updated depending on the contained content components.

Package 1	file: Preliminaries.ppt; contains: Preliminaries;
Package 2	file: DB-Intro.ppt; contains: DB-Introduction;
Package 3	file: DB-ER.ppt; contains: DB-ER-Model;

Package 4	file: DB-RelM.ppt; contains: DB-Relational-Model;
Package 5	file: DB-Design.ppt; contains: DB-Design;
Package 6	file: DB-Theory.ppt; contains: DB-Normalization, DB-Data-Design;
Package 7	file: DB-SQL.ppt; contains: DB-SQL;
Package 8	file: DB-Transactions.ppt; contains: DB-Transactions;
Package 9	file: DB-Recovery.ppt; contains: DB-Recovery;
Package 10	file: DB-DataOrganization.ppt; contains: DB-DataOrganization;
Package 11	file: PetriNets.ppt; contains: PN-Introduction, PN-Transformations, PN-Dynamics, PN-Analysis-Basics, PN-Analysis-Advanced, PN-LinearAlgebra, PN-Extensions;
Package 12	file: WF-Intro.ppt; contains: WF-Introduction;
Package 13	file: WF-ProcMgmt.ppt; contains: WF-ProcessManagement, WF-ProcessModelingIntro;
Package 14	file: WF-Modeling.ppt; contains: WF-Modeling-PetriNets, WF-Modeling-UML, WF-Modeling-EPC;
Package 15	file: WF-MS.ppt; contains: WFMS-Architecture, INCOME-Designer;

- **Requirements for software artifacts**
  - **Helper programs**[2]
    1. *pptappend.exe*: this program is called with two Powerpoint files p_1, p_2 as parameters, and appends the slides of p_2 after the last slide of p_1, and saves the changed version of p_1.
    2. *pptApplyTemplate.exe*: this program is called with two parameters *master*, *p*, where *master* is a Powerpoint master file defining a layout, and *p* a Powerpoint file. The layout of *master* is applied to *p*, and *p* is saved.
    3. *pptApplyTemplateAll.exe*: this program is called with two parameters *master*, *dir*, where *master* is a Powerpoint master file defining a layout and *dir* is a directory that contains Powerpoint files. The program applies *master* on all Powerpoint files it finds in the first directory level, and saves the modified files.
    4. *pptTableOfContent.exe*: this program is called with two parameters *p*, *index*, where *p* is a Powerpoint file and *index* a number indicating a slide number in *p*. The program generates a table of contents (TOC) for *p* based on the headings on each slide and the respective slide numbers. If the TOC fits on one slide, all slide numbers starting from position *index* are increased by one, and a new TOC slide is inserted at position *index*. Similarly, if the TOC consists of more than one slide, the numbers of the slides starting from position *index* are increased by the number of TOC slides, and the new TOC slides are inserted starting at *index*. After the insertion

[2] Remark: 1–6 were wrapper programs that automated Powerpoint tasks in domain engineering, which required MS Powerpoint 2000, XP, or 2003 installed.

of TOC slides, the slide numbers in the TOC are updated, and p is saved with the modifications.

5. *pptTOCall.exe*: this program is called with two parameters *dir*, *index*, where *dir* is a directory and *index* a number indicating a slide number. This program works as *pptTableOfContent.exe*, except that it inserts a table of contents in each Powerpoint file that it finds on the first level of *dir*.
6. *pptToHTML.exe*: this program can be called with two parameters p, h, where p is a Powerpoint file and h an HTML file that will be created by converting p into HTML.
7. *acrobat.exe*: this program can be used in application engineering to convert PPT files to PDF format. This is done through direct user interaction.
8. *showparams.exe*: this program displays on screen all parameters that are passed to it when it is called. It serves testing purposes in domain testing.

Domain Design

Based on the results from domain analysis, the following models were derived in domain design:

- the *conceptual product line model* with the respective domains and features, as depicted in Fig. 10.6.
- the chosen *content component model* was the application-specific model of MS Powerpoint files. All content components in this product line adhered to this model. Metadata was stored in the documents properties section of Powerpoint files.
- *core asset version graph models*: initially, the core asset version graph of each content component had only one node ("v1.0"), as there was only one file for each component; these graphs are therefore omitted from the presentation. For the layout specifications associated to the *Layout* feature (cf. Fig. 10.6), there were three layout versions available, as shown in Fig. 10.7; the layouts differed slightly in the appearance of title slides, logos, and colors.
- the used *product map template* was the database schema presented earlier in Fig. 7.5. The data of the conceptual product line model, the core assets, and the core asset version graph were added in domain realization.
- the *construction specification* was defined by the workflow model in Fig. 10.8.
- *reusable workflow modules*: workflow patterns for the creation of common features and optional features were prepared, similar to Fig. 7.6 a) and b); their repeated presentation is omitted here.

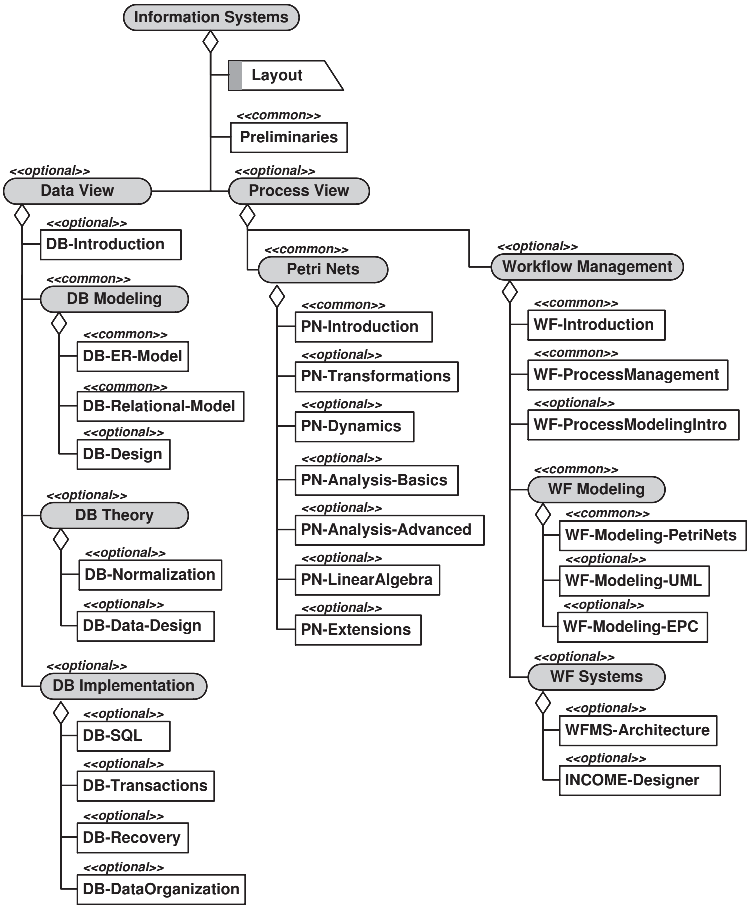


Fig. 10.6. The conceptual product line model.

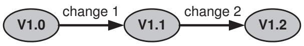


Fig. 10.7. The core asset version graph associated to the layout feature (for details on nodes and edges see Appendix A.2.1).

Domain Realization

In domain realization, the construction artifacts needed for the product line were created. Throughout the case study, a directory structure as shown in Fig. 10.9 was used to organize the various artifacts; it had the subdirectories *DE* for domain engineering, *AE* for application engineering, *FE* for family engineering, and *DWE* for the files of the Desktop Workflow Engine. The *CoreAssets* subdirectory contained various types of core assets, such as documentation, empty package files, helper programs, models (e.g., the conceptual

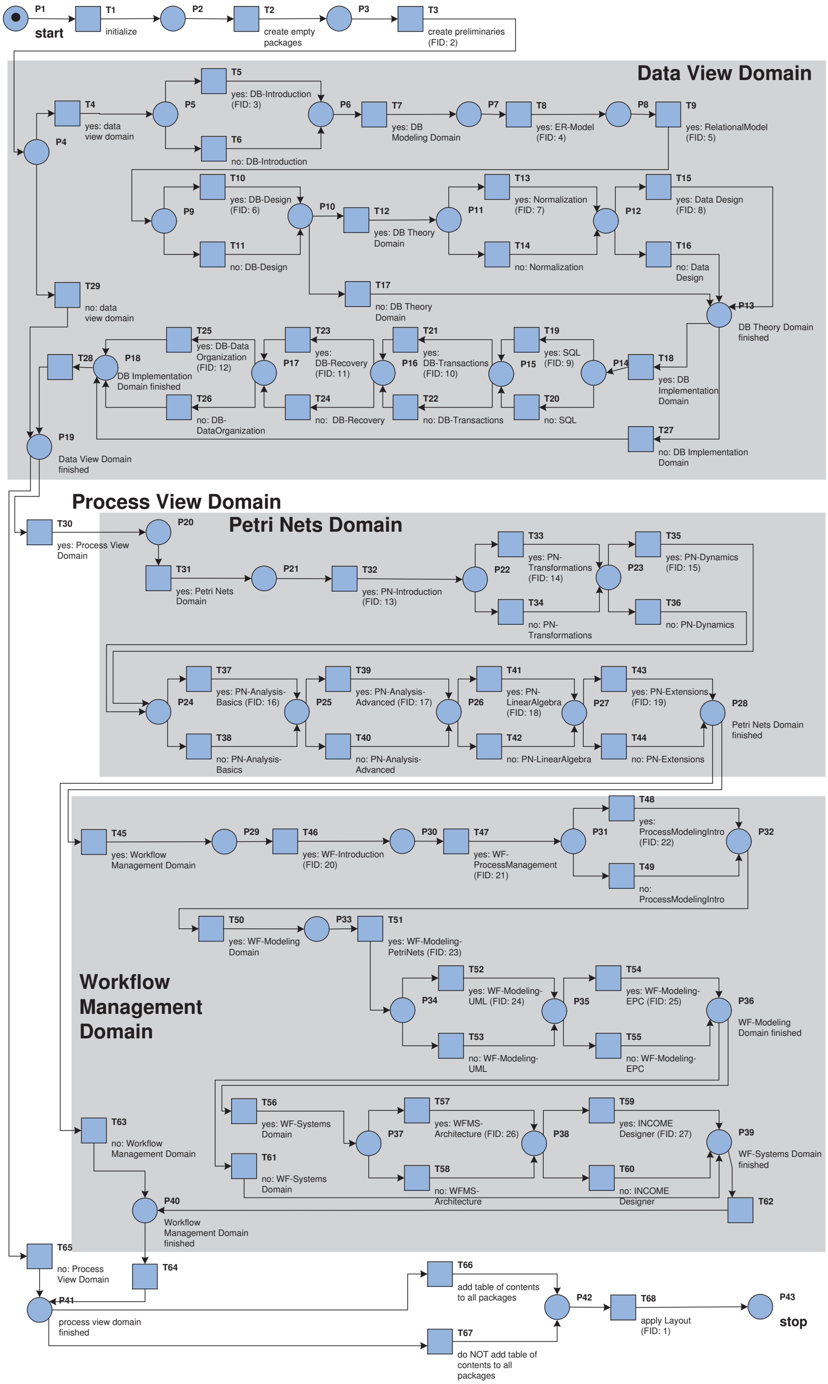
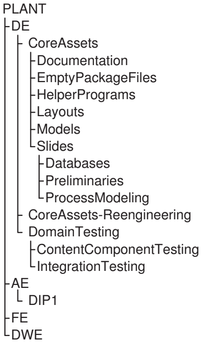

P1
start
T1
initialize
P2
T2
create empty packages
P3
T3
create preliminaries (FID: 2)
Data View Domain
P4
T4
yes: data view domain
P5
T5
yes: DB-Introduction (FID: 3)
T6
no: DB-Introduction
P6
T7
yes: DB Modeling Domain
P7
T8
yes: ER-Model (FID: 4)
P8
T9
yes: RelationalModel (FID: 5)
P9
T10
yes: DB-Design (FID: 6)
T11
no: DB-Design
P10
T12
yes: DB Theory Domain
P11
T13
yes: Normalization (FID: 7)
T14
no: Normalization
P12
T15
yes: Data Design (FID: 8)
T16
no: Data Design
T17
no: DB Theory Domain
P13
DB Theory Domain finished
T29
no: data view domain
T18
yes: DB Implementation Domain
P14
T19
yes: SQL (FID: 9)
T20
no: SQL
P15
T21
yes: DB-Transactions (FID: 10)
T22
no: DB-Transactions
P16
T23
yes: DB-Recovery (FID: 11)
T24
no: DB-Recovery
P17
T25
yes: DB-Data Organization (FID: 12)
T26
no: DB-DataOrganization
P18
DB Implementation Domain finished
T28
T27
no: DB Implementation Domain
P19
Data View Domain finished
Process View Domain
T30
yes: Process View Domain
P20
Petri Nets Domain
T31
yes: Petri Nets Domain
P21
T32
yes: PN-Introduction (FID: 13)
P22
T33
yes: PN-Transformations (FID: 14)
T34
no: PN-Transformations
P23
T35
yes: PN-Dynamics (FID: 15)
T36
no: PN-Dynamics
P24
T37
yes: PN-Analysis-Basics (FID: 16)
T38
no: PN-Analysis-Basics
P25
T39
yes: PN-Analysis-Advanced (FID: 17)
T40
no: PN-Analysis-Advanced
P26
T41
yes: PN-LinearAlgebra (FID: 18)
T42
no: PN-LinearAlgebra
P27
T43
yes: PN-Extensions (FID: 19)
T44
no: PN-Extensions
P28
Petri Nets Domain finished
T45
yes: Workflow Management Domain
P29
T46
yes: WF-Introduction (FID: 20)
P30
T47
yes: WF-ProcessManagement (FID: 21)
P31
T48
yes: ProcessModelingIntro (FID: 22)
T49
no: ProcessModelingIntro
P32
T50
yes: WF-Modeling Domain
P33
T51
yes: WF-Modeling-PetriNets (FID: 23)
Workflow Management Domain
P34
T52
yes: WF-Modeling-UML (FID: 24)
T53
no: WF-Modeling-UML
P35
T54
yes: WF-Modeling-EPC (FID: 25)
T55
no: WF-Modeling-EPC
P36
WF-Modeling Domain finished
T56
yes: WF-Systems Domain
T61
no: WF-Systems Domain
P37
T57
yes: WFMS-Architecture (FID: 26)
T58
no: WFMS-Architecture
P38
T59
yes: INCOME Designer (FID: 27)
T60
no: INCOME Designer
P39
WF-Systems Domain finished
T62
T63
no: Workflow Management Domain
P40
Workflow Management Domain finished
T65
no: Process View Domain
T64
P41
process view domain finished
T66
add table of contents to all packages
T67
do NOT add table of contents to all packages
P42
T68
apply Layout (FID: 1)
P43
stop

Fig. 10.8. The workflow model that represented the construction specification of the product line.

product line model, the workflow model, the SQL DDL statements (cf. [ST05]) to create the database schema representing the product map template), and the slides (which were the content components in this case). The *CoreAssets-Reengineering* subdirectory was used for reengineering of existing material, i.e., to store original material, extracted components, and reengineered content components. The *DomainTesting* subdirectory stored artifacts that were needed for or resulting from testing, and provided a predefined testing environment with various batch files.



```
PLANT
├DE
│ ├ CoreAssets
│ │ ├Documentation
│ │ ├EmptyPackageFiles
│ │ ├HelperPrograms
│ │ ├Layouts
│ │ ├Models
│ │ └Slides
│ │   ├Databases
│ │   ├Preliminaries
│ │   └ProcessModeling
│ ├ CoreAssets-Reengineering
│ └ DomainTesting
│   ├ContentComponentTesting
│   └IntegrationTesting
├AE
│ └ DIP1
├FE
└DWE
```

Fig. 10.9. Outline of the directory structure created to organize the artifacts of PLANT.

- **Realization of content components**
  In the context of this case study, content components were Powerpoint files that covered a topic specified by the atomic features of the conceptual product line model. The content components were created based on material of existing lectures. To create one content component, appropriate slides were extracted from existing files. Thereafter, each component candidate was overworked as follows: the layout was made uniform for all content components, absolute page numbers and chapter numbers in the headlines were deleted, macros and other absolute slide references were deleted, metadata was updated, and the file was saved in one of appropriate subdirectories of the *slides* directory. The content component for *Preliminaries* was created based on existing slides and was a PPT file that contained placeholders for organizational guidelines needed in each lecture; these could be completed later in application engineering with the specific data of a lecture.
- **Realization of other construction artifacts**
  The necessary layout files were realized as Powerpoint master templates. Among others, they contained for example the logo of the institute. The required helper programs were already available from a previous project. The

wrapper programs for Powerpoint were copied to the *HelperPrograms* directory.

- **Realization of the database**
  The database was realized with the database editor of the DWE tool (cf. Fig. 9.1). SQL statements were executed to create the empty database of the product map template, based on Fig. 7.5. Thereafter, the data that was derived from the conceptual product line model, and the core assets and their version graphs was inserted into the tables: *Domain*, *Feature*, *CoreAssetVersionGraph*, *CoreAssetVersion*, *ChangeDescription*, *deriveVersion*, *Package*. For the tables *map* and *DigitalInformation Product* preliminary data was inserted to make domain testing possible; this data could be changed later in application engineering. In addition, other tables were created, which were needed only on the physical level: the table *Log* was created for documentation purposes during workflow execution; the table *Stereotypes* held textual descriptions for stereotype IDs. The views *ViewCPLM*, *ViewDerivedVersions*, *ViewMapDetails* were supplementary and could show different helpful views on the data of the aforementioned tables. The data contained in the database tables is listed in Figures A.1 – A.13 of the Appendix.
- **Realization of the construction workflow model**
  The model depicted in Fig. 10.8 was implemented in the DWE tool. Furthermore, internal variables were specified inside transitions, so that helper programs could be called with parameters obtained from the database. An extract of transitions with details about their internal properties is shown in Appendix A.2.2. The completed workflow model and the associated database were stored with the DWE in one single file, in the DWE-specific format.

Domain Testing

- **Testing the content components**
  For each of the created content components, i.e., Powerpoint file, it was tested whether it was a valid PPT file (by opening it in Powerpoint), whether the predefined layouts could be automatically applied with the helper programs *pptApplyTemplate.exe* and *pptApplyTemplateAll.exe* and whether the files had thereafter the specified layout (with respect to characters, line breaks, text positions, colors, font sizes), whether a table of contents could be correctly generated with *pptTableOfContent.exe* and *pptTOCall.exe*, and whether a PDF/HTML file was correctly produced with *pptToHTML.exe* or *acrobat.exe*. Visual checks assured that no unwanted changes were introduced (according to the criteria mentioned above). This testing procedure has revealed wrong formatting in 4 content components, which were re-designed accordingly. Finally, the content components were

validated by checking whether the content was useful in the way it was created.

- **Verifying the construction workflow model**
  During the implementation of the workflow model in the DWE, the tool checked instantly whether the properties of a workflow net held (according to Def. 7.4). This revealed instantly about 3 design errors (which otherwise would have lead to inconsistent stop markings) during the creation of the model. These errors resulted mainly from the fact that the model was bigger than the visible area on the screen, and could be corrected right away. In addition, the model of the construction workflow was verified to make sure that only desired markings were reachable, which meant that only desired product configurations were realizable when the workflow would be executed. For this, a marking graph was created for the workflow model with the initial marking as shown in Fig. 10.8. The resulting marking graph[3] is depicted in Fig. 10.10. The marking graph had exactly one marking which was dead, which meant that the workflow had a well-defined "stop" state; in addition, the marking graph had finitely many nodes, which meant that the number of tokens was bounded in any place. The marking graph was used to check that only desired markings were reachable from other desired markings, and that a marking could lead only to other desired markings. The marking graph helped to identify and correct 2 control flow errors.

- **Integration testing of content components**
  To test whether the content components could be combined to create information products and whether the database, the workflow, and the called programs produced intended results, 5 sample information products were created. These sample products and their configuration were derived from the marking graph in such a way that each edge in the marking graph was passed (i.e., covered) at least once; a total of 5 different product configurations had to be created, so that every edge in the marking graph was passed at least once. The creation of sample products helped to identify 5 errors during the execution of the workflow, which resulted from wrong or missing execution parameters inside transitions. The sequence of transitions for the creation of these sample products is listed below:
  - Sample product 1:
    T1 … T5, T7 … T10, T12, T13, T15, T18, T19, T21, T23, T25, T28, T30 … T33, T35, T37, T39, T41, T43, T45 … T48, T50 … T52, T54, T56, T57, T59, T62, T64, T66, T68
  - Sample product 2:
    T1 … T3, T29, T65, T67, T68

[3] Remark: The labels of nodes and edges of the marking graph were computed by the DWE tool; the layout of the marking graph was changed for presentation.

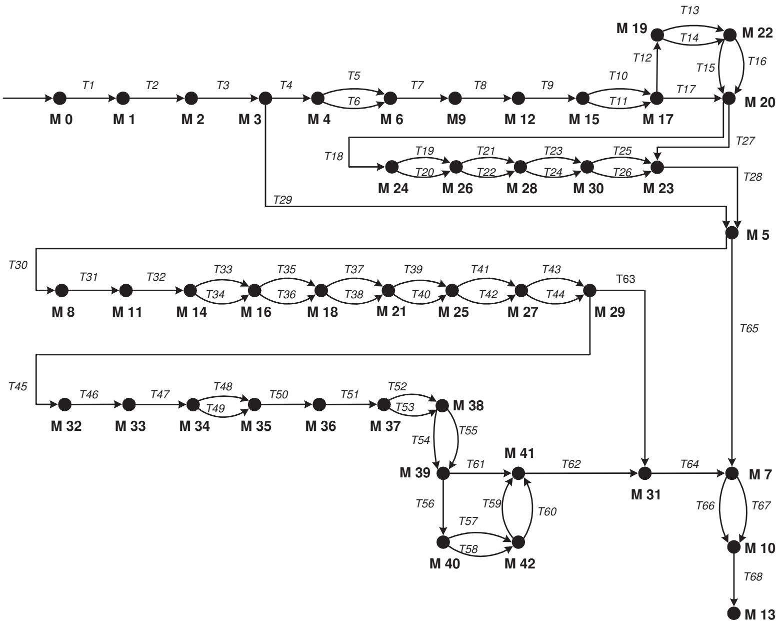


Fig. 10.10. Marking graph for the workflow model in Fig. 10.8.

- Sample product 3:
  T1 . . . T4, T6 . . . T9, T11, T12, T14, T16, T18, T20, T22, T24, T26, T28, T30 . . . T32, T34, T36, T38, T40, T42, T44, T63, T64 ,T66, T68
- Sample product 4:
  T1 . . . T4, T6 . . . T9, T11, T17, T27, T28, T30 . . . T32, T34, T36, T38, T40, T42, T44, T45 . . . T47, T49 . . . T51, T53, T55, T56, T58, T60, T62, T64, T67, T68
- Sample product 5:
  T1 . . . T3, T29, T30 . . . T32, T34, T36, T38, T40, T42, T44, T45 . . . T47, T49, T50, T51, T53, T54, T61, T62, T64, T67, T68

10.5 Application Engineering

This section illustrates how application engineering (see Chapter 8) has been carried out. For the case study, a concrete information product was generated for the course "Applied Informatics 1 (AI-1)". The steps are described next.

Application Analysis

The specific requirements for the AI-1 course were derived from the curriculum.

- Product-specific content requirements
  - the course had to contain a *preliminaries* part with organizational details.
  - the course had to cover the *data view domain* with the *ER model, the relational model, database design, and normalization.*
  - the course had to cover the *process view domain* with an *introduction to Petri nets, transformations on Petri nets, dynamics of Petri nets, and basic analysis techniques for Petri nets.*
  - each package had to have a table of contents.
  - Powerpoint files were required for the lecture in class; for the delivery to students, they had to be converted to PDF files and had to be put on a Web server.

The content requirements were in line with the domain requirements, and were thus realizable with the product line.

Application Design

Based on the requirements from application analysis and the conceptual product line model created during domain engineering (Fig. 10.6), the specific product model was derived for the AI-1 course (Fig. 10.11).

The product map for the AI-1 course is depicted in Fig. 10.12, which shows the data of the respective feature IDs, core asset version IDs, and package IDs that were used in the table *map* of the database that was created in domain engineering (cf. Appendix A.1).

Application Realization

During application realization, the DWE was used to adapt the database created in domain engineering to the requirements of the AI-1 course. In particular, it was ensured that the table *map* contained the IDs specified in application design. Furthermore, the output directory of the product, "C:\PLANT\AE\DIP1" was created, and this directory was ensured to be registered as the base directory for outputs in the table *package* (see Appendix Fig. A.8).

Thereafter, the workflow was executed with the DWE tool to create the AI-1 product. The following sequence of transitions was used to create it (see also Fig. A.14 for details):

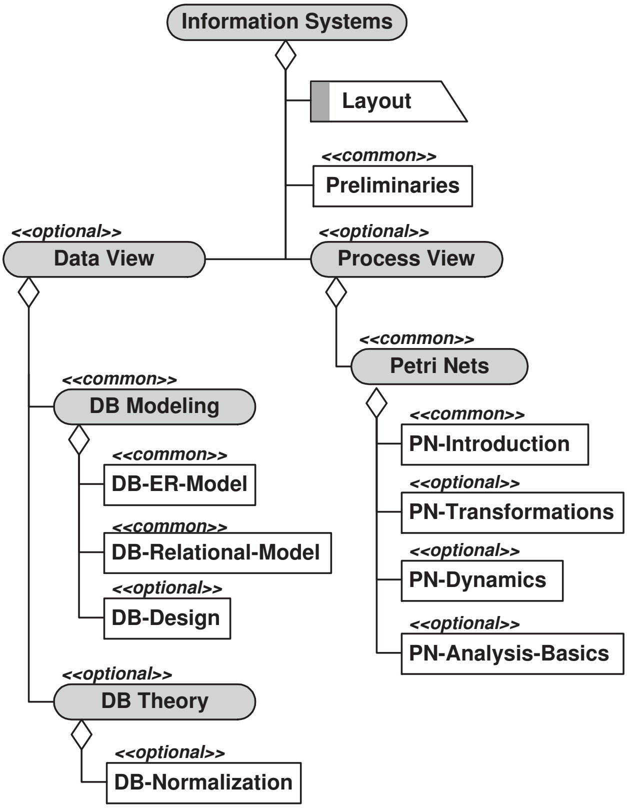
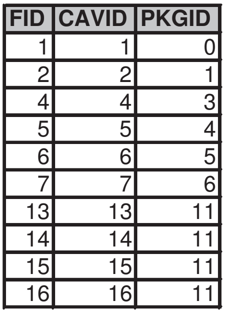


Fig. 10.11. The product model for the AI-1 course.



FID	CAVID	PKGID
1	1	0
2	2	1
4	4	3
5	5	4
6	6	5
7	7	6
13	13	11
14	14	11
15	15	11
16	16	11

Fig. 10.12. The data of the product map for the AI-1 course, which was used in the table *map* of the database (cf. Fig. 7.5). The data of the related tables *feature*, *package*, *coreassetversion* is shown in Appendix A.2.1

- T1 ... T4, T6, T7 ... T10, T12, T13, T16, T27, T28, T30 ... T33, T35, T37, T40, T42, T44, T63, T64, T66, T68.

The DWE produced 6 Powerpoint files in the aforementioned directory, which contained the product-specific content: *Preliminaries.ppt, DB-ER.ppt, DB-RelM.ppt, DB-Design.ppt, DB-Theory.ppt, PetriNets.ppt.* As defined by the workflow model in domain engineering, the content of the feature *DB-ER-Model* was inserted in the file *DB-ER.ppt*, *DB-Relational-Model* in the file *DB-RelM.ppt*, *DB-Design* in the file *DB-Design.ppt*, *DB-Normalization* in the file *DB-Theory.ppt*, and the content of the features *PN-Introduction, PN-Transformations, PN-Dynamics, and PN-Analysis-Basics* were inserted in this sequence in the file *PetriNets.ppt.*

A first iteration of application testing followed, and thereafter another iteration of application realization was done to finish the product-specific tasks that were not automated in the workflow. In particular, the specific organizational details were completed in the file *Preliminaries.ppt*, and product-specific metadata was added to the generated Powerpoint files (e.g., course name, creator, date). Furthermore, the Powerpoint files were converted to PDF. For the PDF files, a second iteration of application testing was done, and back in application realization, the PDF files were completed with metadata and published on the lecture Web site.

Application Testing

In the first iteration of application testing all created Powerpoint files were checked to have a valid file format by opening them in the Powerpoint application and checking them in presentation mode. Visual checks ensured that no unwanted changes were introduced and that existing effects worked as intended. The content and the combination of content was checked to be as specified.

The second iteration of application testing checked the PDF files to have a valid format. Visual checks were performed as well.

10.6 Evaluation

For evaluation of the PLANT Approach, the improvements (in qualitative or quantitative terms) with PLANT are compared to the situation before PLANT was applied (cf. Sect. 10.2).

One main improvement with PLANT was that the number of slides which had to be maintained and updated regularly was reduced. Before PLANT, the courses Applied Informatics I, Workflow Management, Database Systems (at Karlsruhe), Database Systems (at Vienna) had a total of 2197 Powerpoint slides. With PLANT, core assets (i.e., content components, layout specifications) were created out of the existing slides. The conceptual product line model was conceived in such a way that the existing courses could be generated to be with the same content as before, and in addition, other useful variants could be generated as well. The total number of slides of the content components was reduced to 1057. This is due to realized synergy effects, i.e., slides that occurred identically in several products were stored only once in the core asset base, and the DWE was used to insert them in a predefined way into the right products.

Because of these synergy effects, it became possible to improve the updating of content parts that were identical in several information products and make updating easier, because the respective slides could be modified only once in the core asset repository. In addition, the construction specification guaranteed that only products with predefined content configurations could be generated, and that updates of common components were propagated systematically to the respective parts in newly generated products.

PLANT has also brought improvements with respect to the evolution of content and future extensions. Through its models, PLANT made the assumptions about content and the reuse of content explicit, i.e., there was a uniform component model for content in the product line, the composition technique defined within the product line was working with the chosen component model, and the composition language was aligned to the component model and composition technique (cf. Sect. 3.3.4). In this case study, the component model was based on the application-specific format of Powerpoint, the composition technique was realized by wrapper programs that could append slides in one file after slides in another file, and the composition language was defined in principle by the construction specification. If future content components adhered to these assumptions of the product line, they could be integrated more easily to extend the product line.

10.6.1 Possibilities for Generalization of the Results

Having shown that PLANT worked for one product line in the presented context, arguments are now given to explain how PLANT can be applied to create other product lines in other domains, e.g., as introduced in Sect. 5.4.5.

As mentioned in Sect. 5.4.3, PLANT assumes that the following models remain fixed for each product line: the product line metamodel, the core asset version graph model, and the database schema of the product map template. To apply PLANT in a certain domain, it must be possible to express the special requirements of that domain by the conceptual product line model, the construction specification, the content component model, and the construction and test artifacts.

The presented product line used the application-specific content component model of MS Powerpoint, which is represents one of the more difficult cases where the content component model has a binary format and is closely tied to one application. It has been shown that it is possible to construct helper programs that are wrappers around a specific application, which can help to automate the construction of products in a product line. Furthermore, an empirical study [NTD03] has shown that indeed Powerpoint is one of the most frequently used formats for information products in an educational context. Sticking to the application-specific model of Powerpoint for the content components, other product lines can be constructed by re-using the aforementioned helper programs and creating a different conceptual product line model, product map, and an adjusted construction specification.

PLANT is also applicable in areas where the formats of content components are based for example on HTML or XML. The helper programs have to be adapted accordingly to perform the necessary operations (e.g., append content components, apply a layout template, etc.) on the respective file formats, and the helper programs are called as before during the execution of the construction workflow model. The HTML and XML file format are widely used for information products, tools are readily available, or it is feasible to implement the needed tools due to the standardized specifications of these formats (cf. [HM04]). Other content component models and formats can be handled in PLANT only if helper programs can be constructed which automate the operations needed to create information products.

10.7 Summary and Discussion

This case study has shown the applicability of PLANT in a real-life context and demonstrated that PLANT indeed worked in practice. Furthermore, it was documented that for this special case, PLANT has produced measurable improvements of synergy effects by reducing the amount of content that had to be maintained on a regular basis.

11

Conclusion and Outlook

This Chapter summarizes and discusses the results of the thesis, and contrasts the initial situation with the respective deficits to the new contributions and improvements with PLANT. Furthermore, an outlook is given on opportunities for further research.

11.1 New Contributions and Improvements with PLANT

The thesis has shown that digital information products are emerging as a new kind of product, which are becoming more and more important. Due to the specific properties of these products, the creation of variants was shown to be unavoidable, especially when saturation effects in markets have to be compensated.

However, the creation of variants of digital information products is currently not done systematically enough, commonalities are often not exploited, and the success of component reuse is very limited or not there at all, despite the existence of standards. This has been exemplified in detail for the area of e-learning. From a new and more general perspective, it has been worked out that the problems have various sources which are of technical, economic, and organizational nature. Nevertheless, little interdisciplinary work has been done so far to deal with these problems in an integrated way.

The thesis hooks in at this point and introduces a new approach, called Product Lines for Digital Information Products (PLANT), which provides an engineering foundation for the creation of variants of digital information products, and which has different levels, such as strategy, process model, and tools. PLANT unifies and adapts concepts from different fields, e.g., software engineering, information systems, workflow management, databases, economics, or management, in order to tackle the aforementioned problems in a new way that suits the interdisciplinary context.

The main contributions of PLANT are related to the adaptation and extension of concepts from software product lines to fit the new context of digital information products. Similar to software product lines, PLANT makes for information products the tradeoff between standardization and individualization of product variants explicit. As a difference from software product lines, PLANT focuses on the information aspect and the configuration of content in information products, and makes appropriate extensions.

In its process model, PLANT distinguishes between family engineering (with an organizational view), domain engineering (with a view on all information products in one product line), and application engineering (with the view on one single product that can be derived from the artifacts prepared in domain engineering). The models introduced in domain engineering prepare in a systematic and proactive way the production of information product variants in application engineering. In addition, the introduced models make it now possible to give certain guarantees, for example that a produced variant has indeed an allowed content configuration. Moreover, other unique characteristics of PLANT are:

- homogeneous assumptions are imposed on the content components used to build product variants. This ensures that existing components will indeed fit together when information products are built. Furthermore, variants can only contain content from these predefined content components.
- the conceptual product line model gives an overview of possible configurations and the context of reuse of content components. With this model, content developers know from the beginning in which situations certain content is reused. Common content parts are planned from the beginning; this helps to realize synergy effects and also make updates easier. It is now possible to separate crosscutting features (e.g., layout) conceptually as well as technically in the construction process; this simplifies the creation and update of information products for different target groups.
- PLANT has a novel approach to the management of product configurations and parametrization: configuration data (e.g., of the available content components) is stored in a relational database, and the construction workflow model uses this data in an integrated way to automate the creation of an information product. Changes of parameters can be done separately in the database, without having to take all internal details of the workflow model into account. The control flow defined by the construction workflow model can be verified in advance to guarantee that only desired product configurations can be created.
- information products with old configurations can be re-created at any time, as the configuration data is kept in the database and old versions of core assets remain at their specified locations. The data in the database can be systematically analyzed for example for traceability purposes, e.g., to maintain the connections between models, content components, and other artifacts. The database helps to track and proactively control the evolution of information products. The Desktop Workflow Engine realizes

these concepts and represents a solution to what has been identified in [PBvdL05] and Sect. 4 as a research issue.
- PLANT is not limited only to one content format. It can be used also with other file formats (e.g., XML, HTML) and even with binary content formats, such as Powerpoint. The feasibility of the more difficult case with an application-specific, binary format has been demonstrated in the case study.

However, PLANT has also limitations which are incorporated in its assumptions:

- all developed information products must belong to exactly one product line.
- it must be possible to exploit commonalities between similar variants of information products; common content parts must be large enough to justify an investment, and variable parts must be small and predictable.
- it must be often the case that new content of product variants builds upon content already existing in the product line.

The thesis has shown a validation of the proposed approach in a case study, which has also demonstrated the feasibility and advantages of PLANT in a real-life context.

The main challenge was the interdisciplinary nature of the topic of product lines for digital information products. Methods and models from different areas had to be examined, adapted, extended, or combined, e.g., from the areas of software engineering, information systems, workflow management, databases, economics, management, and e-learning. Another particular challenge was the development of the conceptual foundation of PLANT in a way so that it can be used with different content formats (now and in the future) without being bound to a specific format. It was demonstrated that the concepts are applicable even in an extreme (but realistic) case where the content used for products in a product line was available in Powerpoint format, which is a binary and application-specific format that is widely used, and that automation can be achieved also in this case. The Desktop Workflow Engine eases and supports the management of product configurations in a novel way, as well as the integration and usage of configuration data into the construction workflow model.

11.2 Outlook

The thesis provides at several points opportunities for further research, and its interdisciplinary nature also opens new doors for research in related areas:

- at the moment, parametrization in PLANT is supported by the Desktop Workflow Engine which can manage parameter data and pass it on during the construction process of a product. However, parts inside content components are not yet conceived to be parameterizable, since PLANT builds upon existing content component models which are not parameterizable at the moment. Although a specific parameterizable content component model could be created, a more general approach should be investigated in order to avoid compatibility problems. Nevertheless, PLANT is extensible in this respect. A possible direction is to develop a general metalanguage for content parametrization (e.g., similar to [Bas96]) which can be embedded into existing content component models and file formats such as HTML, XML, or Powerpoint, by using a certain syntax and recognizable marker symbols that delimit what belongs to the metalanguage. Then, during construction of an information product, the Desktop Workflow Engine can call appropriate helper programs that parse the appropriate files and insert for example some text or other content in some previously marked places. With such a general solution it would be possible to continue to use widely-spread, as well as application-specific content formats like Powerpoint, and at the same time adapt them more to the product line context.

- traditional workflow management systems are often criticized that due to their centralized architecture, too much supervision of employees is possible at the workplace. The approach taken in this thesis where each developer has an own, local Desktop Workflow Engine under his own control could be investigated further and adapted to suitable business contexts.
- as another extension, it is conceivable to automate further the creation of core assets (especially content components) for the product line of information products in cases where some digital artifacts are already available. For example, existing Web pages, PDF files, or Powerpoint files could be searched to extract the common content that occurs identically in several locations, and create appropriate content components for the common parts and differing parts. In addition, a preliminary conceptual product line model could be created during this process in an automated way.
- more sophisticated automation can enhance the work in various ways. For example, automated analysis and consistency checks can be done between the available models, like the conceptual product line model and the construction workflow model; it is also conceivable to preliminarily generate a construction workflow model out of a given conceptual product line model, which is thereafter modified or extended manually.
- another area deserving more attention is the search and retrieval of workflow modules in a large workflow warehouse, which suit in the context of the currently edited workflow model.
- as information products are often made available on the Internet, a topic for further research would be to use tracking techniques in order to assess which variants are accessed most, and which extensions are probably needed in the future. This way, the evolution of the product line for information products can be controlled in a proactive way. In addition, such

insights would provide additional support in family and domain engineering whether to remove a certain variant and modify the conceptual product line model.

- in domain testing, the integration testing of content components deserves further research for the special case where the marking graph has infinitely many nodes. In such a case, valid product configurations could be obtained with statistical methods. However, it must be assessed how many sample products to create, and which guarantees can be given depending on this number.
- for the e-learning context, the conceptual product line model could be enhanced with additional context-specific data which helps domain and application engineers in a faculty to automatically estimate the effort required for students who enroll in a certain lecture variant, or how many credit points it is worth. It could also be assessed how the conceptual product line model could be used to model a curriculum.
- as a project, a Web-based core asset repository of e-learning material could be created in which all core assets adhere to some product line constraints. The empirical data of the usage of such a repository could provide important insights on the wide-scale usage of product lines for digital information products.
- further research can be done to evaluate QX nets and the Desktop Workflow Engine for the creation of software product lines. In particular, it can be investigated how QX nets should be used to call compilers with parameters in the context of conditional compilation, and to what extent the analysis of the construction workflow model can improve the development of software product lines.
- from a management perspective, organizational change management plays an important role when PLANT is to be introduced. For example, a particular question that needs to be answered is how to (psychologically) motivate people to use the product line approach for digital information products. Also related, details can be worked out when PLANT is to be used in collaborative, inter-organizational scenarios.

A

Appendix

A.1 Implementation Example for the Product Map Template

Relations

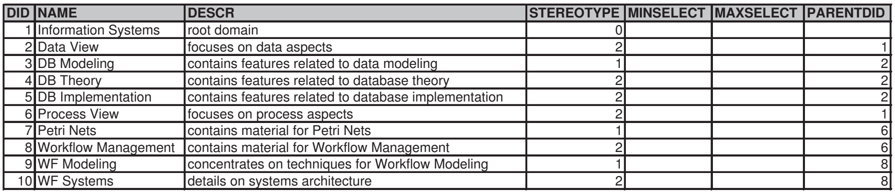
Primary keys are underlined.

E1 $\rightsquigarrow$ **Domain** (dID, name, descr, stereotypeID, minSelect, maxSelect, parentdID)
foreign key parentdID references Domain.dID
E2 $\rightsquigarrow$ **Feature** (fID, name, descr, stereotypeID, dID,cavgID)
foreign key dID references Domain.dID
foreign key cavgID references CoreAssetVersionGraph.cavgID
E3 $\rightsquigarrow$ **CoreAssetVersionGraph** (cavgID, descr)
E4 $\rightsquigarrow$ **CoreAssetVersion** (cavID, name, descr, verNo, filetype, filepath, cavgID)
foreign key cavgID references CoreAssetVersionGraph.cavgID
E5 $\rightsquigarrow$ **ChangeDescription** (cdID, descrtext)
E6 $\rightsquigarrow$ **Package** (pkgID, name, descr, path, dipName)
foreign key dipName references DigitalInformationProduct.Name
E7 $\rightsquigarrow$ **DigitalInformationProduct** (name, creationDate, descr)

R5 $\rightsquigarrow$ **deriveVersion** (childCavID, parentCavID, cdID)
foreign key childCavID references CoreAssetVersion.cavID
foreign key parentCavID references CoreAssetVersion.cavID
foreign key cdID references ChangeDescription.cdID
R6 $\rightsquigarrow$ **map** (fID, cavID, pkgID)
foreign key fID references Feature.fID
foreign key cavID references CoreAssetVersion.cavID
foreign key pkgID references Package.pkgID

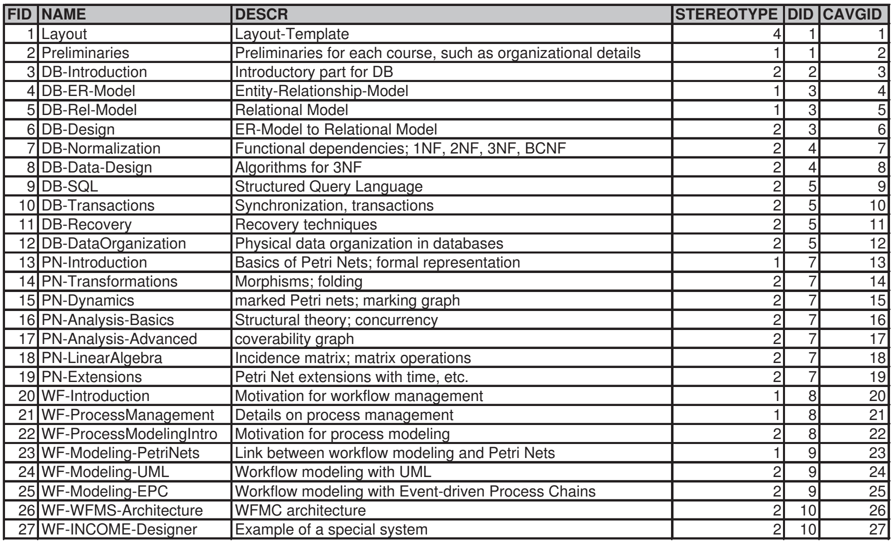
A.2 Case Study – Additional Details

A.2.1 Database used in the Case Study



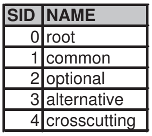
DID	NAME	DESCR	STEREOTYPE	MINSELECT	MAXSELECT	PARENTDID
1	Information Systems	root domain	0			
2	Data View	focuses on data aspects	2			1
3	DB Modeling	contains features related to data modeling	1			2
4	DB Theory	contains features related to database theory	2			2
5	DB Implementation	contains features related to database implementation	2			2
6	Process View	focuses on process aspects	2			1
7	Petri Nets	contains material for Petri Nets	1			6
8	Workflow Management	contains material for Workflow Management	2			6
9	WF Modeling	concentrates on techniques for Workflow Modeling	1			8
10	WF Systems	details on systems architecture	2			8

Fig. A.1. Data in the table *Domain*



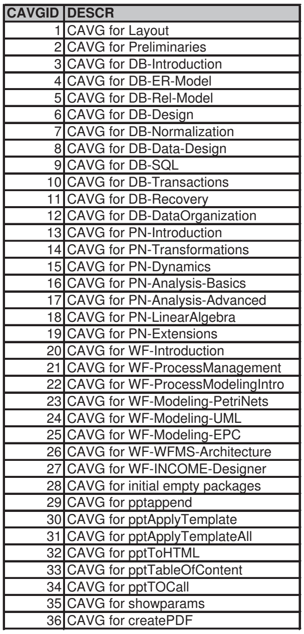
FID	NAME	DESCR	STEREOTYPE	DID	CAVGID
1	Layout	Layout-Template	4	1	1
2	Preliminaries	Preliminaries for each course, such as organizational details	1	1	2
3	DB-Introduction	Introductory part for DB	2	2	3
4	DB-ER-Model	Entity-Relationship-Model	1	3	4
5	DB-Rel-Model	Relational Model	1	3	5
6	DB-Design	ER-Model to Relational Model	2	3	6
7	DB-Normalization	Functional dependencies; 1NF, 2NF, 3NF, BCNF	2	4	7
8	DB-Data-Design	Algorithms for 3NF	2	4	8
9	DB-SQL	Structured Query Language	2	5	9
10	DB-Transactions	Synchronization, transactions	2	5	10
11	DB-Recovery	Recovery techniques	2	5	11
12	DB-DataOrganization	Physical data organization in databases	2	5	12
13	PN-Introduction	Basics of Petri Nets; formal representation	1	7	13
14	PN-Transformations	Morphisms; folding	2	7	14
15	PN-Dynamics	marked Petri nets; marking graph	2	7	15
16	PN-Analysis-Basics	Structural theory; concurrency	2	7	16
17	PN-Analysis-Advanced	coverability graph	2	7	17
18	PN-LinearAlgebra	Incidence matrix; matrix operations	2	7	18
19	PN-Extensions	Petri Net extensions with time, etc.	2	7	19
20	WF-Introduction	Motivation for workflow management	1	8	20
21	WF-ProcessManagement	Details on process management	1	8	21
22	WF-ProcessModelingIntro	Motivation for process modeling	2	8	22
23	WF-Modeling-PetriNets	Link between workflow modeling and Petri Nets	1	9	23
24	WF-Modeling-UML	Workflow modeling with UML	2	9	24
25	WF-Modeling-EPC	Workflow modeling with Event-driven Process Chains	2	9	25
26	WF-WFMS-Architecture	WFMC architecture	2	10	26
27	WF-INCOME-Designer	Example of a special system	2	10	27

Fig. A.2. Data in the table *Feature*



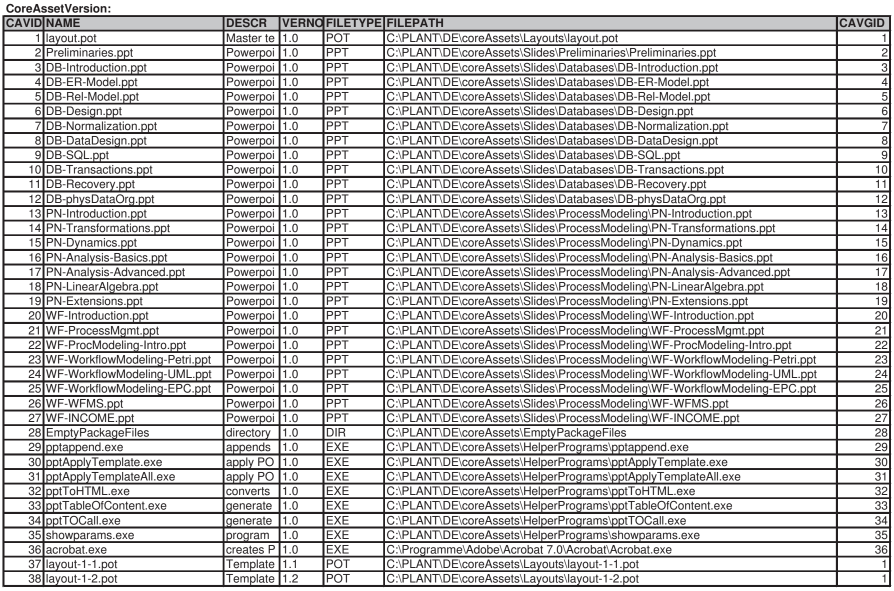
SID	NAME
0	root
1	common
2	optional
3	alternative
4	crosscutting

Fig. A.3. Data in the table *Stereotypes*



CAVGID	DESCR
1	CAVG for Layout
2	CAVG for Preliminaries
3	CAVG for DB-Introduction
4	CAVG for DB-ER-Model
5	CAVG for DB-Rel-Model
6	CAVG for DB-Design
7	CAVG for DB-Normalization
8	CAVG for DB-Data-Design
9	CAVG for DB-SQL
10	CAVG for DB-Transactions
11	CAVG for DB-Recovery
12	CAVG for DB-DataOrganization
13	CAVG for PN-Introduction
14	CAVG for PN-Transformations
15	CAVG for PN-Dynamics
16	CAVG for PN-Analysis-Basics
17	CAVG for PN-Analysis-Advanced
18	CAVG for PN-LinearAlgebra
19	CAVG for PN-Extensions
20	CAVG for WF-Introduction
21	CAVG for WF-ProcessManagement
22	CAVG for WF-ProcessModelingIntro
23	CAVG for WF-Modeling-PetriNets
24	CAVG for WF-Modeling-UML
25	CAVG for WF-Modeling-EPC
26	CAVG for WF-WFMS-Architecture
27	CAVG for WF-INCOME-Designer
28	CAVG for initial empty packages
29	CAVG for pptappend
30	CAVG for pptApplyTemplate
31	CAVG for pptApplyTemplateAll
32	CAVG for pptToHTML
33	CAVG for pptTableOfContent
34	CAVG for pptTOCall
35	CAVG for showparams
36	CAVG for createPDF

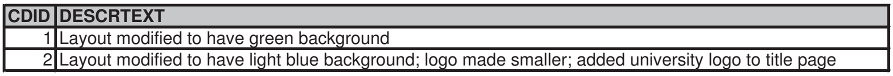
Fig. A.4. Data in the table *CoreAssetVersionGraph*



CoreAssetVersion:

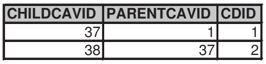
CAVID	NAME	DESCR	VERNO	FILETYPE	FILEPATH	CAVGID
1	layout.pot	Master te	1.0	POT	C:\PLANT\DE\coreAssets\Layouts\layout.pot	1
2	Preliminaries.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\Preliminaries\Preliminaries.ppt	2
3	DB-Introduction.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\Databases\DB-Introduction.ppt	3
4	DB-ER-Model.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\Databases\DB-ER-Model.ppt	4
5	DB-Rel-Model.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\Databases\DB-Rel-Model.ppt	5
6	DB-Design.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\Databases\DB-Design.ppt	6
7	DB-Normalization.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\Databases\DB-Normalization.ppt	7
8	DB-DataDesign.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\Databases\DB-DataDesign.ppt	8
9	DB-SQL.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\Databases\DB-SQL.ppt	9
10	DB-Transactions.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\Databases\DB-Transactions.ppt	10
11	DB-Recovery.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\Databases\DB-Recovery.ppt	11
12	DB-physDataOrg.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\Databases\DB-physDataOrg.ppt	12
13	PN-Introduction.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\ProcessModeling\PN-Introduction.ppt	13
14	PN-Transformations.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\ProcessModeling\PN-Transformations.ppt	14
15	PN-Dynamics.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\ProcessModeling\PN-Dynamics.ppt	15
16	PN-Analysis-Basics.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\ProcessModeling\PN-Analysis-Basics.ppt	16
17	PN-Analysis-Advanced.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\ProcessModeling\PN-Analysis-Advanced.ppt	17
18	PN-LinearAlgebra.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\ProcessModeling\PN-LinearAlgebra.ppt	18
19	PN-Extensions.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\ProcessModeling\PN-Extensions.ppt	19
20	WF-Introduction.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\ProcessModeling\WF-Introduction.ppt	20
21	WF-ProcessMgmt.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\ProcessModeling\WF-ProcessMgmt.ppt	21
22	WF-ProcModeling-Intro.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\ProcessModeling\WF-ProcModeling-Intro.ppt	22
23	WF-WorkflowModeling-Petri.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\ProcessModeling\WF-WorkflowModeling-Petri.ppt	23
24	WF-WorkflowModeling-UML.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\ProcessModeling\WF-WorkflowModeling-UML.ppt	24
25	WF-WorkflowModeling-EPC.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\ProcessModeling\WF-WorkflowModeling-EPC.ppt	25
26	WF-WFMS.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\ProcessModeling\WF-WFMS.ppt	26
27	WF-INCOME.ppt	Powerpoi	1.0	PPT	C:\PLANT\DE\coreAssets\Slides\ProcessModeling\WF-INCOME.ppt	27
28	EmptyPackageFiles	directory	1.0	DIR	C:\PLANT\DE\coreAssets\EmptyPackageFiles	28
29	pptappend.exe	appends	1.0	EXE	C:\PLANT\DE\coreAssets\HelperPrograms\pptappend.exe	29
30	pptApplyTemplate.exe	apply PO	1.0	EXE	C:\PLANT\DE\coreAssets\HelperPrograms\pptApplyTemplate.exe	30
31	pptApplyTemplateAll.exe	apply PO	1.0	EXE	C:\PLANT\DE\coreAssets\HelperPrograms\pptApplyTemplateAll.exe	31
32	pptToHTML.exe	converts	1.0	EXE	C:\PLANT\DE\coreAssets\HelperPrograms\pptToHTML.exe	32
33	pptTableOfContent.exe	generate	1.0	EXE	C:\PLANT\DE\coreAssets\HelperPrograms\pptTableOfContent.exe	33
34	pptTOCall.exe	generate	1.0	EXE	C:\PLANT\DE\coreAssets\HelperPrograms\pptTOCall.exe	34
35	showparams.exe	program	1.0	EXE	C:\PLANT\DE\coreAssets\HelperPrograms\showparams.exe	35
36	acrobat.exe	creates P	1.0	EXE	C:\Programme\Adobe\Acrobat 7.0\Acrobat\Acrobat.exe	36
37	layout-1-1.pot	Template	1.1	POT	C:\PLANT\DE\coreAssets\Layouts\layout-1-1.pot	1
38	layout-1-2.pot	Template	1.2	POT	C:\PLANT\DE\coreAssets\Layouts\layout-1-2.pot	1

Fig. A.5. Data in the table *CoreAssetVersion*



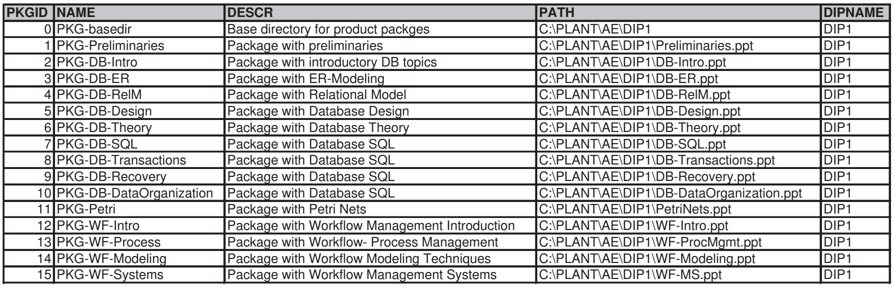
CDID	DESCRTEXT
1	Layout modified to have green background
2	Layout modified to have light blue background; logo made smaller; added university logo to title page

Fig. A.6. Data in the table *ChangeDescription*



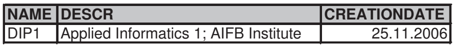
CHILDCAVID	PARENTCAVID	CDID
37	1	1
38	37	2

Fig. A.7. Data in the table *DeriveVersion*



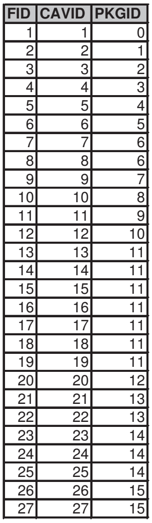
PKGID	NAME	DESCR	PATH	DIPNAME
0	PKG-basedir	Base directory for product packges	C:\PLANT\AE\DIP1	DIP1
1	PKG-Preliminaries	Package with preliminaries	C:\PLANT\AE\DIP1\Preliminaries.ppt	DIP1
2	PKG-DB-Intro	Package with introductory DB topics	C:\PLANT\AE\DIP1\DB-Intro.ppt	DIP1
3	PKG-DB-ER	Package with ER-Modeling	C:\PLANT\AE\DIP1\DB-ER.ppt	DIP1
4	PKG-DB-RelM	Package with Relational Model	C:\PLANT\AE\DIP1\DB-RelM.ppt	DIP1
5	PKG-DB-Design	Package with Database Design	C:\PLANT\AE\DIP1\DB-Design.ppt	DIP1
6	PKG-DB-Theory	Package with Database Theory	C:\PLANT\AE\DIP1\DB-Theory.ppt	DIP1
7	PKG-DB-SQL	Package with Database SQL	C:\PLANT\AE\DIP1\DB-SQL.ppt	DIP1
8	PKG-DB-Transactions	Package with Database SQL	C:\PLANT\AE\DIP1\DB-Transactions.ppt	DIP1
9	PKG-DB-Recovery	Package with Database SQL	C:\PLANT\AE\DIP1\DB-Recovery.ppt	DIP1
10	PKG-DB-DataOrganization	Package with Database SQL	C:\PLANT\AE\DIP1\DB-DataOrganization.ppt	DIP1
11	PKG-Petri	Package with Petri Nets	C:\PLANT\AE\DIP1\PetriNets.ppt	DIP1
12	PKG-WF-Intro	Package with Workflow Management Introduction	C:\PLANT\AE\DIP1\WF-Intro.ppt	DIP1
13	PKG-WF-Process	Package with Workflow- Process Management	C:\PLANT\AE\DIP1\WF-ProcMgmt.ppt	DIP1
14	PKG-WF-Modeling	Package with Workflow Modeling Techniques	C:\PLANT\AE\DIP1\WF-Modeling.ppt	DIP1
15	PKG-WF-Systems	Package with Workflow Management Systems	C:\PLANT\AE\DIP1\WF-MS.ppt	DIP1

Fig. A.8. Data in the table *Package*



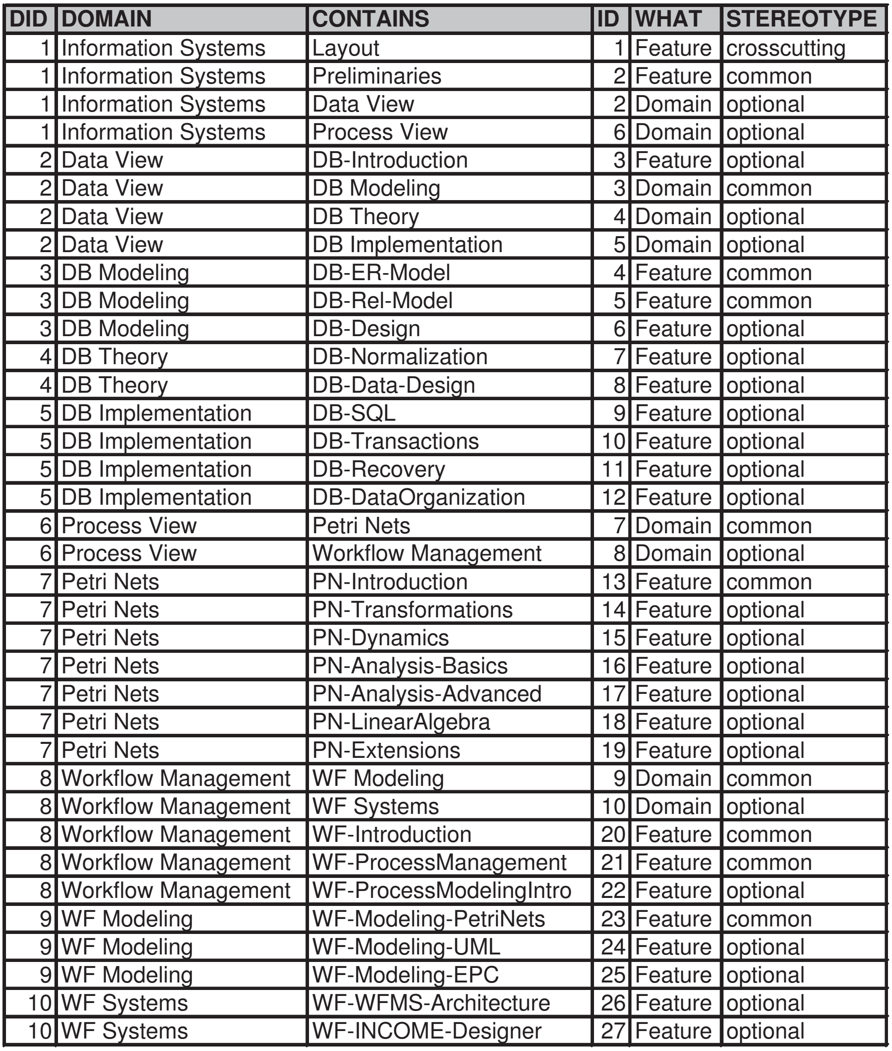
NAME	DESCR	CREATIONDATE
DIP1	Applied Informatics 1; AIFB Institute	25.11.2006

Fig. A.9. Data in the table *DigitalInformationProduct*



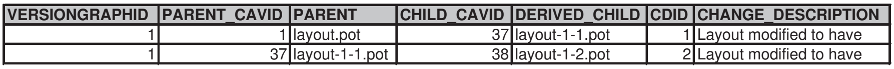
FID	CAVID	PKGID
1	1	0
2	2	1
3	3	2
4	4	3
5	5	4
6	6	5
7	7	6
8	8	6
9	9	7
10	10	8
11	11	9
12	12	10
13	13	11
14	14	11
15	15	11
16	16	11
17	17	11
18	18	11
19	19	11
20	20	12
21	21	13
22	22	13
23	23	14
24	24	14
25	25	14
26	26	15
27	27	15

Fig. A.10. Data in the table *Map*



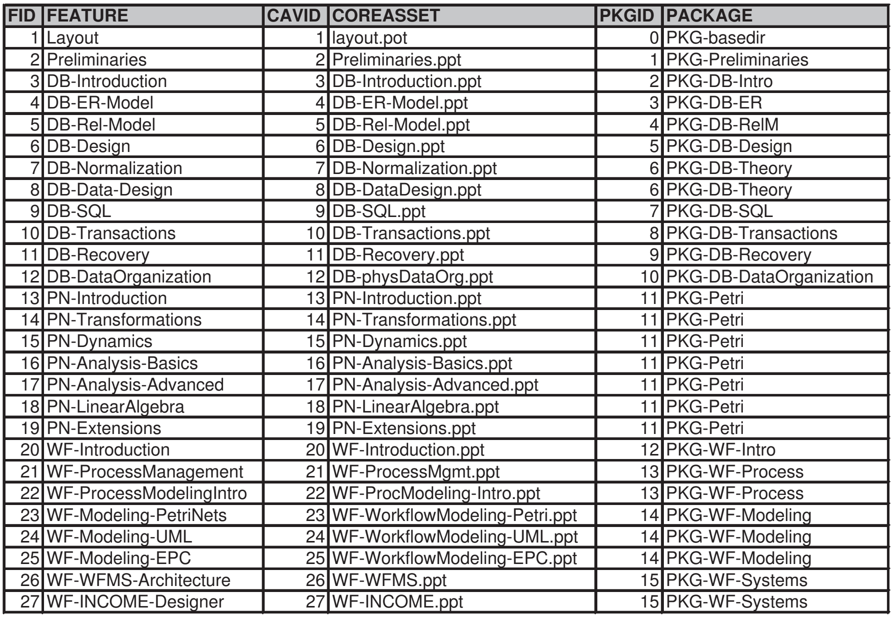
DID	DOMAIN	CONTAINS	ID	WHAT	STEREOTYPE
1	Information Systems	Layout	1	Feature	crosscutting
1	Information Systems	Preliminaries	2	Feature	common
1	Information Systems	Data View	2	Domain	optional
1	Information Systems	Process View	6	Domain	optional
2	Data View	DB-Introduction	3	Feature	optional
2	Data View	DB Modeling	3	Domain	common
2	Data View	DB Theory	4	Domain	optional
2	Data View	DB Implementation	5	Domain	optional
3	DB Modeling	DB-ER-Model	4	Feature	common
3	DB Modeling	DB-Rel-Model	5	Feature	common
3	DB Modeling	DB-Design	6	Feature	optional
4	DB Theory	DB-Normalization	7	Feature	optional
4	DB Theory	DB-Data-Design	8	Feature	optional
5	DB Implementation	DB-SQL	9	Feature	optional
5	DB Implementation	DB-Transactions	10	Feature	optional
5	DB Implementation	DB-Recovery	11	Feature	optional
5	DB Implementation	DB-DataOrganization	12	Feature	optional
6	Process View	Petri Nets	7	Domain	common
6	Process View	Workflow Management	8	Domain	optional
7	Petri Nets	PN-Introduction	13	Feature	common
7	Petri Nets	PN-Transformations	14	Feature	optional
7	Petri Nets	PN-Dynamics	15	Feature	optional
7	Petri Nets	PN-Analysis-Basics	16	Feature	optional
7	Petri Nets	PN-Analysis-Advanced	17	Feature	optional
7	Petri Nets	PN-LinearAlgebra	18	Feature	optional
7	Petri Nets	PN-Extensions	19	Feature	optional
8	Workflow Management	WF Modeling	9	Domain	common
8	Workflow Management	WF Systems	10	Domain	optional
8	Workflow Management	WF-Introduction	20	Feature	common
8	Workflow Management	WF-ProcessManagement	21	Feature	common
8	Workflow Management	WF-ProcessModelingIntro	22	Feature	optional
9	WF Modeling	WF-Modeling-PetriNets	23	Feature	common
9	WF Modeling	WF-Modeling-UML	24	Feature	optional
9	WF Modeling	WF-Modeling-EPC	25	Feature	optional
10	WF Systems	WF-WFMS-Architecture	26	Feature	optional
10	WF Systems	WF-INCOME-Designer	27	Feature	optional

Fig. A.11. Data in the view *ViewCPLM*



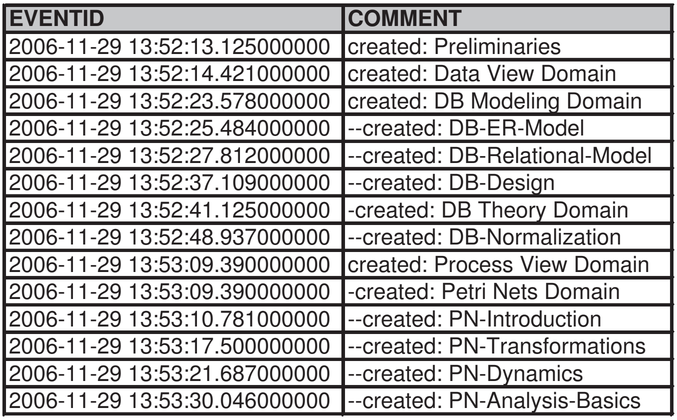
VERSIONGRAPHID	PARENT_CAVID	PARENT	CHILD_CAVID	DERIVED_CHILD	CDID	CHANGE_DESCRIPTION
1	1	layout.pot	37	layout-1-1.pot	1	Layout modified to have
1	37	layout-1-1.pot	38	layout-1-2.pot	2	Layout modified to have

Fig. A.12. Data in the view *ViewDerivedVersions*



FID	FEATURE	CAVID	COREASSET	PKGID	PACKAGE
1	Layout	1	layout.pot	0	PKG-basedir
2	Preliminaries	2	Preliminaries.ppt	1	PKG-Preliminaries
3	DB-Introduction	3	DB-Introduction.ppt	2	PKG-DB-Intro
4	DB-ER-Model	4	DB-ER-Model.ppt	3	PKG-DB-ER
5	DB-Rel-Model	5	DB-Rel-Model.ppt	4	PKG-DB-RelM
6	DB-Design	6	DB-Design.ppt	5	PKG-DB-Design
7	DB-Normalization	7	DB-Normalization.ppt	6	PKG-DB-Theory
8	DB-Data-Design	8	DB-DataDesign.ppt	6	PKG-DB-Theory
9	DB-SQL	9	DB-SQL.ppt	7	PKG-DB-SQL
10	DB-Transactions	10	DB-Transactions.ppt	8	PKG-DB-Transactions
11	DB-Recovery	11	DB-Recovery.ppt	9	PKG-DB-Recovery
12	DB-DataOrganization	12	DB-physDataOrg.ppt	10	PKG-DB-DataOrganization
13	PN-Introduction	13	PN-Introduction.ppt	11	PKG-Petri
14	PN-Transformations	14	PN-Transformations.ppt	11	PKG-Petri
15	PN-Dynamics	15	PN-Dynamics.ppt	11	PKG-Petri
16	PN-Analysis-Basics	16	PN-Analysis-Basics.ppt	11	PKG-Petri
17	PN-Analysis-Advanced	17	PN-Analysis-Advanced.ppt	11	PKG-Petri
18	PN-LinearAlgebra	18	PN-LinearAlgebra.ppt	11	PKG-Petri
19	PN-Extensions	19	PN-Extensions.ppt	11	PKG-Petri
20	WF-Introduction	20	WF-Introduction.ppt	12	PKG-WF-Intro
21	WF-ProcessManagement	21	WF-ProcessMgmt.ppt	13	PKG-WF-Process
22	WF-ProcessModelingIntro	22	WF-ProcModeling-Intro.ppt	13	PKG-WF-Process
23	WF-Modeling-PetriNets	23	WF-WorkflowModeling-Petri.ppt	14	PKG-WF-Modeling
24	WF-Modeling-UML	24	WF-WorkflowModeling-UML.ppt	14	PKG-WF-Modeling
25	WF-Modeling-EPC	25	WF-WorkflowModeling-EPC.ppt	14	PKG-WF-Modeling
26	WF-WFMS-Architecture	26	WF-WFMS.ppt	15	PKG-WF-Systems
27	WF-INCOME-Designer	27	WF-INCOME.ppt	15	PKG-WF-Systems

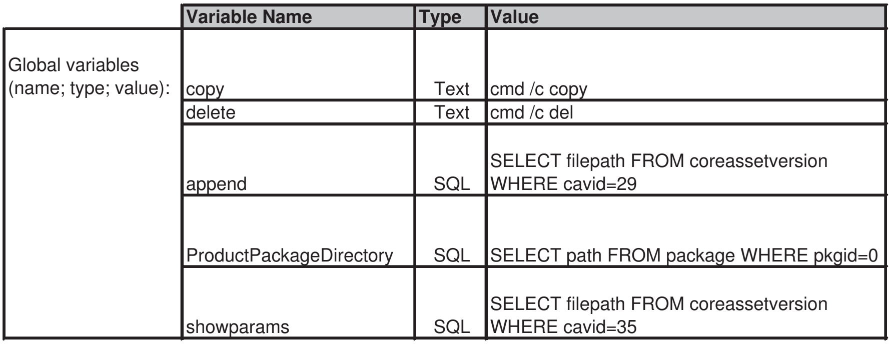
Fig. A.13. Data in the view *ViewMapDetails*



EVENTID	COMMENT
2006-11-29 13:52:13.125000000	created: Preliminaries
2006-11-29 13:52:14.421000000	created: Data View Domain
2006-11-29 13:52:23.578000000	created: DB Modeling Domain
2006-11-29 13:52:25.484000000	--created: DB-ER-Model
2006-11-29 13:52:27.812000000	--created: DB-Relational-Model
2006-11-29 13:52:37.109000000	--created: DB-Design
2006-11-29 13:52:41.125000000	-created: DB Theory Domain
2006-11-29 13:52:48.937000000	--created: DB-Normalization
2006-11-29 13:53:09.390000000	created: Process View Domain
2006-11-29 13:53:09.390000000	-created: Petri Nets Domain
2006-11-29 13:53:10.781000000	--created: PN-Introduction
2006-11-29 13:53:17.500000000	--created: PN-Transformations
2006-11-29 13:53:21.687000000	--created: PN-Dynamics
2006-11-29 13:53:30.046000000	--created: PN-Analysis-Basics

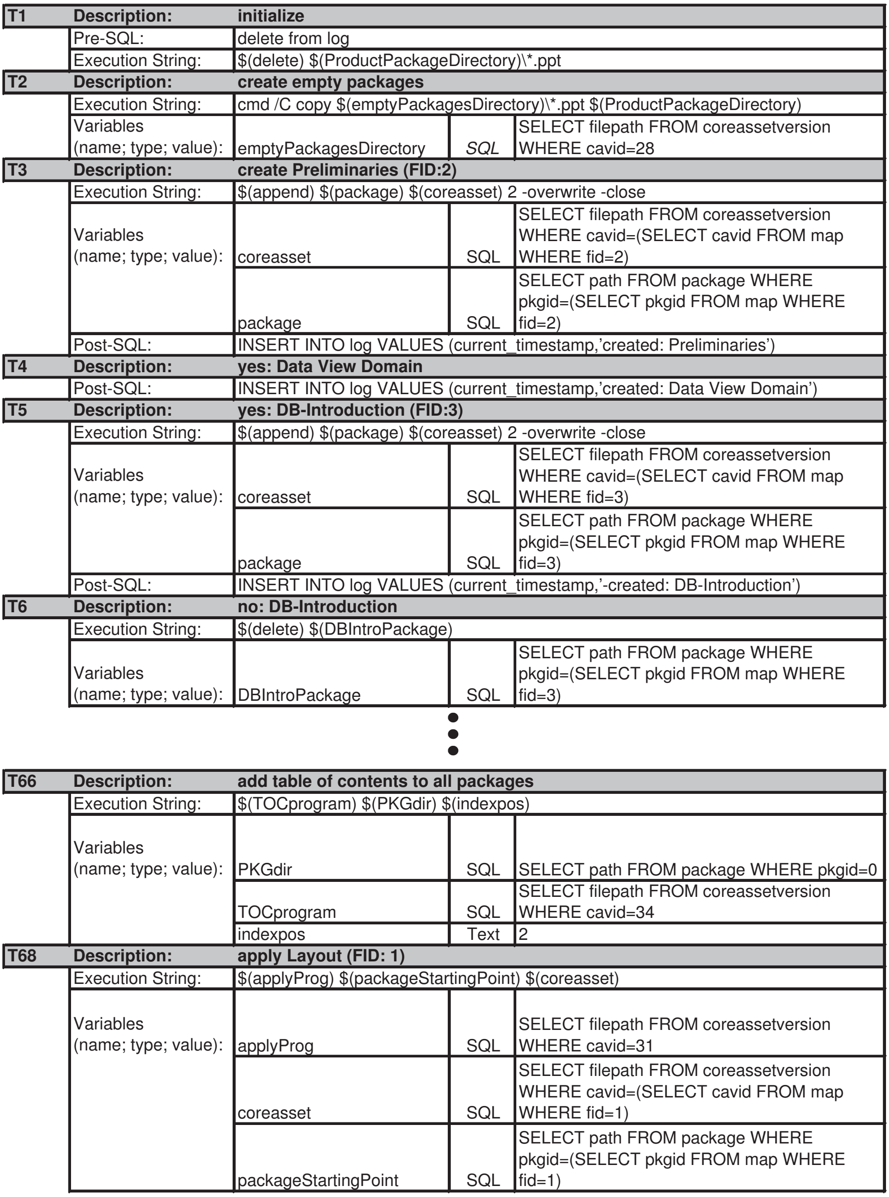
Fig. A.14. Data in the table *Log*, after the generation of the product with the product model in Fig. 10.11

A.2.2 Internals of the Workflow Model



	Variable Name	Type	Value
Global variables (name; type; value):	copy	Text	cmd /c copy
	delete	Text	cmd /c del
	append	SQL	SELECT filepath FROM coreassetversion WHERE cavid=29
	ProductPackageDirectory	SQL	SELECT path FROM package WHERE pkgid=0
	showparams	SQL	SELECT filepath FROM coreassetversion WHERE cavid=35

Fig. A.15. Global variables accessible in all transitions.



T1	**Description:**	**initialize**		
	Pre-SQL:	delete from log		
	Execution String:	$(delete) $(ProductPackageDirectory)*.ppt		
T2	**Description:**	**create empty packages**		
	Execution String:	cmd /C copy $(emptyPackagesDirectory)*.ppt $(ProductPackageDirectory)		
	Variables (name; type; value):	emptyPackagesDirectory	*SQL*	SELECT filepath FROM coreassetversion WHERE cavid=28
T3	**Description:**	**create Preliminaries (FID:2)**		
	Execution String:	$(append) $(package) $(coreasset) 2 -overwrite -close		
	Variables (name; type; value):	coreasset	SQL	SELECT filepath FROM coreassetversion WHERE cavid=(SELECT cavid FROM map WHERE fid=2)
		package	SQL	SELECT path FROM package WHERE pkgid=(SELECT pkgid FROM map WHERE fid=2)
	Post-SQL:	INSERT INTO log VALUES (current_timestamp,'created: Preliminaries')		
T4	**Description:**	**yes: Data View Domain**		
	Post-SQL:	INSERT INTO log VALUES (current_timestamp,'created: Data View Domain')		
T5	**Description:**	**yes: DB-Introduction (FID:3)**		
	Execution String:	$(append) $(package) $(coreasset) 2 -overwrite -close		
	Variables (name; type; value):	coreasset	SQL	SELECT filepath FROM coreassetversion WHERE cavid=(SELECT cavid FROM map WHERE fid=3)
		package	SQL	SELECT path FROM package WHERE pkgid=(SELECT pkgid FROM map WHERE fid=3)
	Post-SQL:	INSERT INTO log VALUES (current_timestamp,'-created: DB-Introduction')		
T6	**Description:**	**no: DB-Introduction**		
	Execution String:	$(delete) $(DBIntroPackage)		
	Variables (name; type; value):	DBIntroPackage	SQL	SELECT path FROM package WHERE pkgid=(SELECT pkgid FROM map WHERE fid=3)

⋮

T66	**Description:**	**add table of contents to all packages**		
	Execution String:	$(TOCprogram) $(PKGdir) $(indexpos)		
	Variables (name; type; value):	PKGdir	SQL	SELECT path FROM package WHERE pkgid=0
		TOCprogram	SQL	SELECT filepath FROM coreassetversion WHERE cavid=34
		indexpos	Text	2
T68	**Description:**	**apply Layout (FID: 1)**		
	Execution String:	$(applyProg) $(packageStartingPoint) $(coreasset)		
	Variables (name; type; value):	applyProg	SQL	SELECT filepath FROM coreassetversion WHERE cavid=31
		coreasset	SQL	SELECT filepath FROM coreassetversion WHERE cavid=(SELECT cavid FROM map WHERE fid=1)
		packageStartingPoint	SQL	SELECT path FROM package WHERE pkgid=(SELECT pkgid FROM map WHERE fid=1)

Fig. A.16. An extract of the transitions with local properties.

References

[AB05] J. Andersson, J. Bosch. Development and use of dynamic product-line architectures. *IEE Proceedings-Software*, 152(1):13–26, 2005.

[ACP01] H. H. Adelsberger, B. Collis, J. M. Pawlowski. *Handbook on Information Technologies for Education and Training.* International Handbooks on Information Systems. Springer Verlag, 2001.

[ADH$^+$00] M. Ardis, N. Daley, D. Hoffman, H. Siy, D. Weiss. Software product lines: a case study. *Software: Practice and Experience*, 30(7):825–847, 2000.

[Ado05] Adobe. PDF reference, fifth edition: Adobe portable document format version 1.6. Technical report, Adobe Systems Incorporated, 2005.

[AG01] M. Anastasopoulos, C. Gacek. Implementing product line variabilities. In *Proceedings of the 2001 Symposium on Software Reusability (SSR '01)*, pages 109–117, New York, NY, USA, 2001. ACM Press.

[AH02a] W. Aalst, K. Hee. *Workflow Management. Models, Methods, and Systems.* MIT Press, 2002.

[AH02b] W. Aalst, A. Hofstede. Workflow patterns: On the expressive power of (petri-net-based) workflow languages. In K. Jensen, editor, *Proceedings of the Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002)*, volume 560 of *DAIMI*, pages 1–20, Aarhus, Denmark, August 2002. University of Aarhus.

[ALSU07] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, second edition, 2007.

[AM05] A. Abran, J. W. Moore, editors. *Guide to the Software Engineering Body of Knowledge (SWEBOK ®).* IEEE Computer Society Professional Practices Committee, February 16 2005. http://www.swebok.org.

[AN03] U. Aßmann, R. Neumann. Quo vadis Komponentensysteme? Von Modulen zu grauen Komponenten. *HMD Praxis der Wirtschaftsinformatik*, 231:19–27, 2003.

[Ara94] G. Arango. Domain analysis. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 424–434. Wiley, 1994.

[ARI07] ARIADNE. ARIADNE Foundation. http://www.ariadne-eu.org, January 2007.

[Aßm03] U. Aßmann. *Invasive Software Composition.* Springer Verlag, 2003.

[Ate04] K. Ateyeh. *Reuse-Driven Courseware Engineering*. PhD thesis, Universität Fridericiana zu Karlsruhe (TH), Germany, 2004.

[Aud07] Audiowalk. The audible city walk. http://www.audiowalk.net, January 2007.

[Avi07] Aviation Industry CBT Committee (AICC). AICC guidelines and reccommendations. http://www.aicc.org, January 2007.

[Bal00] H. Balzert. *Lehrbuch der Software-Technik. Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum Akademischer Verlag, second edition, 2000.

[Bas96] P. G. Bassett. *Framing Software Reuse. Lessons from the real world*. Yourdon Press Computing Series, 1996.

[Bat05] D. S. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines: 9th International Conference (SPLC 2005)*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer Verlag, 2005.

[Bau96] B. Baumgarten. *Petri-Netze. Grundlagen und Anwendungen*. Spektrum Akademischer Verlag, 1996.

[Bau05] P. Baumgartner. Wie man das richtige Content Management Tool für ein bestimmtes Lernmodell auswählt. http://www.elearningeuropa.info, May 17 2005.

[BBRC06] D. Batory, D. Benavides, A. Ruiz-Cortes. Automated analysis of feature models: challenges ahead. *Commun. ACM*, 49(12):45–47, 2006.

[BBSS01] A. Back, O. Bendel, D. Stoller-Schai. *E-Learning im Unternehmen. Grundlagen - Strategien - Methoden - Technologien*. Orell Füssli Verlag, 2001.

[BBZ04] H. Balzert, H. Balzert, O. Zwintzscher. Die E-Learning-Plattform W3L - Anforderungen, Didaktik, Ergonomie, Architektur, Entwicklung, Einsatz. *Wirtschaftsinformatik*, 46(2):129–138, 2004.

[BCM$^+$04] G. Böckle, P. Clements, J. D. McGregor, D. Muthig, K. Schmid. Calculating ROI for software product lines. *IEEE Software*, 21(3):23–31, 2004.

[BCR02] V. R. Basili, G. Caldiera, D. H. Rombach. Goal Question Metric (GQM) Approach. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 578–583. Wiley, second edition, 2002. revision by Rini van Solingen.

[Bec04] M. Becker. *Anpassungsunterstützung in Software-Produktfamilien*. PhD thesis, Technische Universität Kaiserslautern, Kaiserslautern, Germany, 2004.

[BFK$^+$99] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, J.-M. DeBaud. Pulse: a methodology to develop software product lines. In *Proceedings of the 1999 symposium on Software reusability (SSR '99)*, pages 122–131, New York, NY, USA, 1999. ACM Press.

[BHST04] Y. Bontemps, P. Heymans, P.-Y. Schobbens, J.-C. Trigaux. The semantics of foda feature diagrams. In *Workshop on Software Variability Management for Product Derivation*, Boston, MA, August 2004. Helsinki University of Technology.

[BIT05] BITKOM. *Daten zur Informationsgesellschaft*. BITKOM, 2005.

[BIT06] BITKOM. *Daten zur Informationsgesellschaft*. BITKOM, 2006.

[BK98] P. Buxmann, W. König. Das Standardisierungsproblem: Zur ökonomischen Auswahl von Standards in Informationssystemen. *Wirtschaftsinformatik*, 40(2):122–129, 1998.

[BKPS04] G. Böckle, P. Knauber, K. Pohl, K. Schmid. *Software-Produktlinien.* Dpunkt Verlag, 2004.

[BL01] C. Barritt, D. Lewis. Reusable learning object strategy. definition, creation process, and guidelines for building. Technical report, Cisco Systems, Inc., April 22 2001. Version 3.1.

[BMB04] *Kursbuch eLearning 2004. Produkte aus dem Förderprogramm.* Bundesministerium für Bildung und Forschung, 2004.

[BMM99] R. A. Brealey, S. C. Myers, A. J. Marcus. *Fundamentals of Corporate Finance.* McGraw-Hill/Irwin, 1999.

[Boe81] B. W. Boehm. *Software Engineering Economics.* Prentice Hall, 1981.

[Boe88] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[Bos00] J. Bosch. *Design and Use of Software Architectures.* Addison-Wesley, 2000.

[Bos01] J. Bosch. Software product lines: Organizational alternatives. In *23rd International Conference on Software Engineering (ICSE2001)*, pages 91–100, 2001.

[Bos02] J. Bosch. Maturity and evolution in software product lines: Approaches, artefacts and organization. In *Software Product Lines: Second International Conference (SPLC 2)*, volume 2379 of *Lecture Notes in Computer Science*, pages 257–271. Springer Verlag, January 2002.

[Bos03] N. Boskic. Learning objects design: what do educators think about the quality and reusability of learning objects? In *Proceedings. The 3rd IEEE International Conference on Advanced Learning Technologies*, pages 306–307, 9–1 July 2003.

[Bro92] *Brockhaus Enzyklopädie, 19. Auflage., Band 17: Pes-Rac.* Brockhaus, Mannheim, 1992.

[Bro95] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, 20th anniversary edition, 1995.

[Bro99] K. Brockhoff. *Produktpolitik.* UTB, Stuttgart, 1999.

[BS73] F. Black, M. Scholes. The pricing of options and corporate liabilities. *The Journal of Political Economy*, 81(3):637–654, 1973.

[Bun04] Bundesministerium des Inneren. V-Modell® XT. http://www.v-modell-xt.de, 2004.

[BW95] R. Benjamin, R. Wigand. Electronic markets and virtual value chains on the information superhighway. *Sloan Management Review*, 36(2):62–72, Winter95 1995.

[CA04] J.-A. Christiansen, T. Anderson. Feasibility of course development based on learning objects: Research analysis of three case studies. *International Journal of Instructional Technology and Distance Education*, 1(3), April 2004.

[Can04] CanCore. CanCore metadata standard. http://www.cancore.ca/guidelines/CanCore_Guidelines_Introduction_2.0.pdf, April 2004.

[CAR07] CAREO. Campus Alberta Repository of Educational Objects (CAREO). http://careo.netera.ca, January 2007.

[CD02] F. Casati, U. Dayal, editors. *Special Issue on Web Services*, volume 25. IEEE Bulletin of the Technical Committee on Data Engineering, December 2002.

[CE00] K. Czarnecki, U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CEN07] CEN. CEN/ISSS learning technologies workshop. http://www.cenorm.be/cenorm/index.htm, January 2007.

[CHE04] K. Czarnecki, S. Helsen, U. W. Eisenecker. Staged configuration using feature models. In *Software Product Lines: Third International Conference (SPLC 2004)*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283. Springer Verlag, 2004.

[CI90] E. Chikofsky, J. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[Cif94] C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, School of Computing Science, July 1994.

[CKK06] K. Czarnecki, C. H. P. Kim, K. T. Kalleberg. Feature models are views on ontologies. In *Software Product Lines: 10th International Conference (SPLC 2006)*, pages 41–51, 2006.

[Cle96] P. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, 22-23 March 1996.

[CM02] G. Chastek, J. D. McGregor. Guidelines for developing a product line production plan. Technical report, Software Engineering Institute (SEI), Carnegie Mellon University, June 2002. CMU/SEI-2002-TR-06, ESC-TR-2002-006.

[CMS05] CMS Works, Inc. The CMS report. http://www.cmswatch.com/CMS/Report, Summer 2005.

[CN02] P. Clements, L. M. Northrop. *Software Product Lines: Practices and Patterns*. The SEI Series in Software Engineering. Addison-Wesley, August 20 2002.

[Coc05] T. Cochrane. *Podcasting: Do It Yourself Guide*. Wiley, 2005.

[Coh03] S. Cohen. Predicting when product line investment pays. Technical Report CMU/SEI-2003-TN-017, Software Engineering Institute (SEI), Carnegie Mellon University, 2003.

[Cre06] Creole Eclipse Plugin. http://www.thechiselgroup.org/creole, December 2 2006.

[CSFP04] B. Collins-Sussman, B. W. Fitzpatrick, C. M. Pilato. *Version Control with Subversion*. O'Reilly, 2004.

[CSW97] S.-Y. Choi, D. O. Stahl, A. B. Whinston. *The Economics of Electronic Commerce*. MacMillan Publishing Company, 1997.

[CW85] L. Cardelli, P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, 1985.

[CW98] R. Conradi, B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[dAFGD02] R. de Almeida Falbo, G. Guizzardi, K. C. Duarte. An ontological approach to domain engineering. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering (SEKE '02)*, pages 351–358, New York, NY, USA, 2002. ACM Press.

[Dan05] Danish Technological Institute. Study of the e-learning suppliers “market” in europe. Technical report, Danish Technological Institute, Independent consultant Jane Massy, Alphametrics Ltd, Heriot-Watt University, January 2005.

[DDL01] N. Dholakia, R. R. Dholakia, M. Laub. Electronic commerce and the transformation of marketing. In *Global E-Commerce and Online Marketing: Watching the Evolution*, page 43. Quorum/Greenwood, 2001.

[DEH$^+$00] P. Dourish, W. K. Edwards, J. Howell, A. LaMarca, J. Lamping, K. Petersen, M. Salisbury, D. Terry, J. Thornton. A programming model for active documents. In *UIST '00: Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 41–50, New York, NY, USA, 2000. ACM Press.

[Des07] DescribeThis. Automatic parser and generator of dublin core metadata for online resources. http://www.describethis.com/, January 2007.

[DH03] E. Duval, W. Hodgins. A LOM research agenda. In *Proceedings of the 12th international conference on World Wide Web*, pages 1–9, 2003.

[Dij72] E. W. Dijkstra. Notes on structured programming. In O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare, editors, *Structured Programming*. Academic Press, 1972.

[DK73] M. Darby, E. Karni. Free competition and the optimal amount of fraud. *Journal of Law & Economics*, 16(1):67–88, 1973.

[DO96] J. Desel, A. Oberweis. Petri-Netze in der Angewandten Informatik. *Wirtschaftsinformatik*, 38(4):359–367, 1996.

[Doc07] Official homepage for DocBook: The Definitive Guide. http://docbook.org, January 2007.

[Dod02] M. H. Dodani. The dark side of object learning: Learning objects. *Journal of Object Technology*, 1(5):37–42, 2002. ETH Zürich.

[Dow03] S. Downes. Design and reusability of learning objects in an academic context: A new economy of education? *USDLA Journal*, 17(1), January 2003.

[Dow04] S. Downes. The rise of learning objects. *International Journal of Instructional Technology and Distance Education*, 1(3), April 2004.

[DOZZ97] J. Desel, A. Oberweis, T. Zimmer, G. Zimmermann. Validation of information system models: Petri nets and test case generation. In *IEEE International Conference on Systems, Man, and Cybernetics*, pages 3401–3406, 1997.

[DR98] J. Desel, W. Reisig. Place/transition petri nets. In *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*, pages 122–173. Springer Verlag, June 1998.

[DS99] J.-M. DeBaud, K. Schmid. A systematic approach to derive the scope of software product lines. In *Proceedings of the 21st international conference on Software engineering (ICSE)*, pages 34–43, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[DSB04] S. Deelstra, M. Sinnema, J. Bosch. Experiences in software product families: Problems and issues during product derivation. In *Software Product Lines: Third International Conference (SPLC 2004)*, volume 3145 of *Lecture Notes in Computer Science*, pages 165–182. Springer Verlag, 2004.

[DT06a] P. Dodds, S. E. Thropp. Advanced Distributed Learning (ADL) Sharable Content Object Reference Model (SCORM)® 2004. Content Aggregation Model (CAM). http://www.adlnet.org, November 16 2006. 3rd edition.

[DT06b] P. Dodds, S. E. Thropp. Advanced Distributed Learning (ADL) Sharable Content Object Reference Model (SCORM)® 2004. http://www.adlnet.org, November 16 2006. 3rd edition.

[DTA05] O. Díaz, S. Trujillo, F. I. Anfurrutia. Supporting production strategies as refinements of the production process. In *Software Product Lines: 9th International Conference (SPLC 2005)*, volume 3714 of *Lecture Notes in Computer Science*, pages 210–221. Springer Verlag, 2005.

[Duv04] E. Duval. Learning technology standardization: Making sense of it all. *Computer Science and Information Systems*, 1(1):33–43, February 2004. ComSIS Consortium.

[EdN06] EdNA. EdNA (Education Network Australia) metadata standard. http://www.edna.edu.au/edna/go/resources/metadata, October 2006.

[EdN07] EdNA. Education Network Australia (EdNA online). http://www.edna.edu.au, January 2007.

[EEL$^+$02] J. Elliott, R. Eckstein, M. Loy, D. Wood, B. Cole. *Java Swing*. O'Reilly, 2nd edition, 2002.

[EER02] Industry profit margins improve as educational tech. companies strive for viable business models. Electronic Education Report, Vol. 9(6), 3/26 2002. p. 1-3.

[EIT03] EITO. *European Information Technology Observatory*. European Economic Interest Grouping, 2003.

[EIT05] EITO. *European Information Technology Observatory*. European Economic Interest Grouping, 2005.

[eMa05] eMarketer. Reports online publishing: Focus on newspapers. www.emarketer.com, August 2005.

[ESK04] M. El-Sherbini, G. Klim. Metadata and cataloging practices. *The Electronic Library*, 22(3):238, 2004. Emerald.

[ESSW01] D. Ehrenberg, A.-W. Scheer, M. Schumann, U. Winand. Implementierung von interuniversitären Lehr- und Lernkooperationen: Das Beispiel WINFOLine. *Wirtschaftsinformatik*, 43(1):5–11, 2001.

[Eur97] European Commission. *Strategic developments for the European publishing industry towards the year 2000: Europe's multimedia challenge*. European Commission, DG XIII-E, Luxembourg, 1997. EUO-OP Reference: CD-09-97-001-EN-C, Euroabstract Number: 35/533.

[Eur04] *Report on the consultation workshops "Access Rights for e-Learning Content" & "Creating, sharing and reusing e-Learning Content"*, Brussels, October 27–28 2004. European Commission. Directorate-General for Education and Culture.

[FECA04] R. Filman, T. Elrad, S. Clarke, M. Aksit. *Aspect-Oriented Software Development*. Addison-Wesley, October 6 2004.

[Fei91] P. H. Feiler. Configuration management models in commercial environments. Technical Report CMU/SEI-91-TR-7, Carnegie Mellon University, March 1991.

[FFF98] J. M. Favaro, K. R. Favaro, P. F. Favaro. Value based software reuse investment. *Annals of Software Engineering*, 5(0):5–52, January 1998.

[FLT04] R. G. Farrell, S. D. Liburd, J. C. Thomas. Dynamic assembly of learning objects. In *Alternate track papers & posters of the 13th international conference on World Wide Web*, pages 162–169. ACM Press, 2004.

[Fri04a] N. Friesen. Final report on the “international LOM survey”. Technical report, ISO/IEC JTC1 SC36 N0871, September 8 2004.

[Fri04b] N. Friesen. Three objections to learning objects. In *R. McGreal (Ed.). Online Education Using Learning Objects*. London: Routledge/Falmer, 2004.

[FS86] J. Farrell, G. Saloner. Installed base and compatibility: Innovation, product preannouncements, and predation. *The American Economic Review*, 76(5):940–955, 1986.

[Gab04] *Gabler Wirtschaftslexikon*. Dr. Th. Gabler Verlag, 2004.

[GFd98] M. Griss, J. Favaro, M. d’Alessandro. Integrating feature modeling with the RSEB. In *5th International Conference on Software Reuse*, pages 76–85, Vancouver, BC, Canada, June 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GM05] R. J. Glushko, T. McGrath. *Document engineering: analyzing and designing documents for business informatics & Web services*. MIT Press, 2005.

[Gom05] H. Gomaa. *Designing Software Product Lines with UML*. Addison-Wesley, 2005.

[Goo07] Google. Google Earth - A 3D interface to the planet. http://earth.google.com, January 2007.

[GSC$^+$04] J. Greenfield, K. Short, S. Cook, S. Kent, J. Crupi. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[GSMK04] K. Götzer, U. Schneiderath, B. Maier, T. Komke. *Dokumenten-Management*. Dpunkt Verlag, 2004.

[GSNHS03] A. Geyer-Schulz, A. Neumann, A. Heitmann, K. Stroborn. Strategic positioning options for scientific libraries in markets of scientific and technical information - the economic impact of digitization. *The Journal of Digital Information*, 4(2):Article No. 168, 2003-05-09 2003.

[Gun78] R. C. Gunther. *Management methodology for software product engineering*. Wiley, 1978.

[HC00] W. Hodgins, M. Conner. Everything you ever wanted to know about learning standards but were afraid to ask. http://www.linezine.com /2.1/features/wheyewtkls.htm, Fall 2000.

[Hil03] M. R. Hilbert. *From Industrial Economics to Digital Economics: An Introduction to the Transition: Productive Development*. Number 100 in Series Productive Development. United Nations Publications, 2003.

[HK03] A. Hettrich, N. Korolova. Marktstudie Learning Management Systeme (LMS) und Learning Content Management Systeme (LCMS). Fokus deutscher Markt. Technical report, Fraunhofer Institut für Arbeitswirtschaft und Organisation IAO, June 2003.

[HM04] E. R. Harold, W. S. Means. *XML in a Nutshell*. O’Reilly, third edition, 2004.

[Hoe06] T. Hoeren. Skriptum Internetrecht. http://www.uni-muenster.de/Jura.itm/hoeren/material/Skript/skript_Juni2006.pdf, June 2006.

[Hol95] D. Hollingsworth. Workflow management coalition. the workflow reference model. WFMC-TC-1003, January 19 1995.

[HP03] G. Halmans, K. Pohl. Communicating the variability of a software-product family to customers. *Software and Systems Modeling*, 2(1):15–36, March 2003. Springer Verlag.

[HREW04] M. Hatala, G. Richards, T. Eap, J. Willms. The interoperability of learning object repositories and services: standards, implementations and lessons learned. In *Alternate track papers & posters of the 13th international conference on World Wide Web*, pages 19–27. ACM Press, 2004.

[HSI05] G. Q. Huang, T. W. Simpson, B. J. P. II. The power of product platforms in mass customisation. *International Journal of Mass Customisation (IJMASSC)*, 1(1):1–13, 2005.

[Hum02] W. S. Humphrey. Personal software process. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 2, pages 949–961. Wiley, 2nd edition, 2002.

[HWV96] M. Hämäläinen, A. B. Whinston, S. Vishik. Electronic markets for learning: education brokerages on the internet. *Communications of the ACM*, 39(6):51–58, 1996.

[IBK$^+$06] H. P. In, J. Baik, S. Kim, Y. Yang, B. Boehm. A quality-based cost estimation model for the product line life cycle. *Commun. ACM*, 49(12):85–88, 2006.

[IBM07] IBM. Rational Unified Process. http://www-306.ibm.com/software/awdtools/rup/, January 2007.

[IEE90] IEEE. Standard glossary of software engineering terminology. IEEE Std 610.12-1990, September 28 1990.

[IEE02] IEEE Computer Society. IEEE Standard for Learning Object Metadata. http://standards.ieee.org, IEEE Std P1484.12.1$^{\mathrm{TM}}$-2002, September 6 2002.

[IEE05] IEEE LTSC. IEEE Learning Technology Standards Committee. http://ltsc.ieee.org, December 2005.

[IMS04] IMS Global Learning Consortium, Inc. IMS content packaging information model, version 1.1.4. http://www.imsglobal.org/content/packaging/index.html, October 2004.

[IMS07] IMS Global Learning Consortium, Inc. IMS Global Learning Consortium, Inc. - Specifications. http://www.imsglobal.org/specifications.html, January 2007.

[Inm96] W. Inmon. *Building the Data Warehouse*. Wiley, 2nd edition, 1996.

[Int07] Intute. Web resources for education and research. http://www.intute.ac.uk, January 2007.

[ISO96] ISO/IEC. Information technology - Syntactic metalanguage - Extended BNF. International Standard. http://standards.iso.org/ittf/PubliclyAvailableStandards, December 15 1996. ISO/IEC 14977:1996.

[ISO03a] ISO. Information and documentation – The Dublin Core metadata element set. ISO 15836:2003(E), February 26 2003.

[ISO03b] ISO/IEC. Introduction of the core elements set in localized LOM model. http://mdlet.jtc1sc36.org/doc/SC36_WG4_N0059.pdf, September 9 2003. ISO/IEC JTC1 SC36 working document.

[ISO05] ISO. ISO/IEC JTC1 SC36. Standards for: Information Technology for Learning, Education, and Training (ITLET). http://jtc1sc36.org/, December 2005.

[Iss02] L. J. Issing. Instruktions-Design für Multimedia. In L. J. Issing, P. Klimsa, editors, *Information und Lernen mit Multimedia und Internet*. Verlagsgruppe Beltz, 2002.

[Jak06] Jakarta Commons. http://jakarta.apache.org/commons, September 2006.

[JF88] R. Johnson, B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming (JOOP)*, 1(2):22–35, June/July 1988.

[JGJ97] I. Jacobson, M. Griss, P. Jonsson. *Software reuse: architecture, process and organization for business success*. ACM Press/Addison-Wesley, 1997.

[Jon04] R. Jones. Designing adaptable learning resources with learning object patterns. *Journal of Digital Information*, 6(1):Article No. 305, 2004.

[JUN06] Java Universal Network/Graph Framework (JUNG). http://jung.sourceforge.net, December 2006.

[KA05] P. Kotler, G. Armstrong. *Principles of Marketing*. Prentice Hall, 11th edition, 2005.

[KCHP90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. N. A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, ESD-90-TR-222, Software Engineering Institute (SEI), Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1990.

[KD05] H. Kosch, M. Döller. MPEG: Überblick und Integration in Multimedia-Datenbanken. *Datenbank-Spektrum*, 15:36–43, 2005.

[KD06] T. Käkölä, J. C. Duenas, editors. *Software Product Lines. Research Issues in Engineering and Management*. Springer Verlag, 2006.

[KL05] E. Kaplan-Leiserson. American Society for Training and Development (ASTD). Learning Circuits - Glossary. http://www.learningcircuits.org/glossary, November 2005.

[KLD02] K. Kang, J. Lee, P. Donohoe. Feature-oriented product line engineering. *IEEE Software*, 19(4):58–65, 2002.

[Kle02] M. Klein. *Courseware Engineering – ein Vorgehensmodell zur Erstellung von wiederverwendbaren, hypermedialen Kursen*. PhD thesis, University of Karlsruhe, 2002.

[KN06] T. Kishi, N. Noda. Formal verification and software product lines. *Commun. ACM*, 49(12):73–77, 2006.

[Kno04] G. Knolmayer. E-Learning Objects. *Wirtschaftsinformatik*, 46(3):222–224, 2004.

[Knu68] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Correction: Mathematical Systems Theory 5(1): 95-96(1971).

[Koo92] Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung. *Vorgehensmodell*, volume 27. Bundesanzeiger, August 1992.

[KR00] R. Kalakota, M. Robinson. *e-Business 2.0*. Addison-Wesley, 2000.

[KRS04] H.-B. Kittlaus, C. Rau, J. Schulz. *Software-Produkt-Management*. Springer Verlag, 2004.

[Kru92] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[KS00] J. Kuusela, J. Savolainen. Requirements engineering for product families. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 61–69, New York, NY, USA, 2000. ACM Press.

[KT02] G. Kazakevich, L. Torlina. Consumer choice, information product quality, and market implications. Technical report, Deakin University. School of Information Systems. Working Paper 2002/61, 2002.

[KWB03] A. Kleppe, J. Warmer, W. Bast. *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley, April 2003.

[L'A97] J. J. L'Allier. A frame of reference: NETg's map to its products, their structures and core beliefs, 1997.

[Leo01] U. Leonhardt. *Digitales Produkt - Beispiel einer Integrationsplattform für Technik- und Verkaufsprozesse mittels Informations- und Visualisierungstechnologien*. PhD thesis, ETH Zürich, 2001.

[Lev81] T. Levitt. Marketing intangible products and product intangibles. *Harvard Business Review*, 59(3):94–102, May/June 1981.

[Lik32] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 140, 1932.

[Lip76] R. Lipton. The reachability problem requires exponential space. Technical Report YALE/DCS/TR063, Dept. Computer Science, 1976.

[LL03] J. Laudon, K. Laudon. *Management Information Systems*. Prentice Hall, 8th international edition, 2003.

[LMQS03] E. Law, K. Maillet, J. Quemada, B. Simon. Educanext: A service for knowledge sharing. In *Proceedings of the 3rd Annual Ariadne Conference*. ARIADNE Foundation, 2003.

[Loe00] C. Loebbecke. Online delivered content - the core of the intangible economy. In S. Barnes, B. Hunt, editors, *E-Commerce and V-Business: Business Models for Global Success*. Elsevier, 2000.

[LOG06] Apache Log4j Project. http://logging.apache.org/log4j, September 2006.

[Lou02] K. C. Louden. *Programming Languages: Principles and Practice*. Thomson Brooks/Cole, second edition, 2002.

[LR99] F. Leymann, D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall, September 1999.

[LVS$^+$03] P. Lyman, H. R. Varian, K. Swearingen, P. Charles, N. Good, L. L. Jordan, J. Pal. How much information 2003? http://www.sims.berkeley.edu/research/projects/how-much-info-2003, 2003. reviewed January 2007.

[Mac03] Macromedia. Macromedia Flash (SWF) file format specification, version 7. Technical report, Macromedia, 2003.

[May84] E. Mayr. An algorithm for the general petri net reachability problem. *SIAM Journal on Computing*, 13(3):441–460, 1984.

[McG01] J. D. McGregor. Testing a software product line. Technical Report CMU/SEI-2001-TR-022, Software Engineering Institute (SEI), Carnegie Mellon University, 2001.

[McG04] J. D. McGregor. Product production. *Journal of Object Technology*, 3(10):89–98, November/December 2004. ETH Zürich.

[McI68] M. McIlroy. Mass-produced software components. In *Proceedings of the NATO Conference on Software Engineering*, Garmisch, Germany, 1968.

[Mec04] R. Mecklenburg. *Managing Projects with GNU Make (Nutshell Handbooks)*. O'Reilly, 2004.

[Mef00] H. Meffert. *Marketing*. Gabler Verlag, 2000.

[Mer02] M. Merz. *E-Commerce und E-Business*. Dpunkt Verlag, 2002.

[MER07] MERLOT. Multimedia educational resource for learning and online teaching (MERLOT). http://www.merlot.org, January 2007.

[MHS05] M. Mernik, J. Heering, A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[Mic07] Microsoft. Microsoft office online home page. http://office.microsoft.com, January 2007.

[Mor00] J. Moran. Top ten e-learning myths. *Training & Development*, 54(9):32, 2000.

[MS98] M. H. Meyer, R. Seliger. Product platforms in software development. *Sloan Management Review*, 40(1):61–74, Fall 1998.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, volume 77, pages 541–580, 1989.

[Mut03] D. Muthig. *A light-Weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines*. PhD thesis, University of Kaiserslautern, 2003. Fraunhofer IRB-Verlag.

[MYB89] T. W. Malone, J. Yates, R. I. Benjamin. The Logic of Electronic Markets. Executive Summary. *Harvard Business Review*, 67(3):ES24, May/June 1989.

[MZ96] M. H. Meyer, M. H. Zack. The design and development of information products. *Sloan Management Review*, 37(3):43–59, Spring 1996.

[NDTN03] J. Najjar, E. Duval, S. Ternier, F. Neven. Towards interoperable learning object repositories: the ARIADNE experience. In *Proceedings of the IADIS International Conference WWW/Internet 2003*, volume 1, pages 219–226, Algarve, Portugal, 5-8 November 2003. IADIS.

[Nei80] J. M. Neighbors. *Software Construction Using Components*. PhD thesis, University of California at Irvine, 1980. ICS-TR-160.

[Nel70] P. Nelson. Information and consumer behavior. *The Journal of Political Economy*, 78(2):311–329, Mar.-Apr. 1970.

[Ng01] S. Ng. Learning resource identification. http://www.itsc.org.sg/standards_news/2001-10/SamuelNg-LRI.pdf, October 2001.

[NHHMA03] H. Niegemann, S. Hessel, D. Hochscheid-Mauel, K. Aslanski. *Kompendium E-Learning*. Springer Verlag, 2003.

[NM05] L. Neal, D. Miller. The basics of e-learning: an excerpt from handbook of human factors in web design. *ACM eLearn*, 2005(8):2, 2005.

[NTD03] J. Najjar, S. Ternier, E. Duval. The actual use of metadata in ARIADNE: an empirical analysis. In *Proceedings of the 3rd Annual ARIADNE Conference*, pages 1–6. ARIADNE Foundation, 2003.

[Obj07] Object Management Group (OMG). Unified modeling language. http://www.uml.org, January 2007.

[OPS05] A. Oberweis, V. Pankratius, W. Stucky. Product lines in e-learning. Technical Report 501, Institute of Applied Informatics and Formal Description Methods, University of Karlsruhe, Germany, August 2005.

[OPS06] A. Oberweis, V. Pankratius, W. Stucky. Product lines for digital information products. *Information Systems*, 2006. accepted: Sept. 28, 2006; available online Nov. 2, 2006; in press.

[OS96] A. Oberweis, P. Sander. Information system behavior specification by high level petri nets. *ACM Transactions on Information Systems*, 14(4):380–420, 1996.

[Oxf98] *The Oxford Dictionary for International Business*. Oxford University Press, 1998. Compiled by Market House Books Ltd.

[PA01] J. M. Pawlowski, H. H. Adelsberger. Standardisierung von Lerntechnologien. *Wirtschaftsinformatik*, 43(1):57–68, 2001.

[Pan05] V. Pankratius. Aspect-oriented learning objects. In *4th IASTED International Conference on Web-based Education*, Grindelwald, Switzerland, February 2005. ACTA Press.

[Par01] D. L. Parnas. On the design and development of program families. In D. M. Hoffmann, D. M. Weiss, editors, *Software Fundamentals. Collected Papers by David L. Parnas*, pages 193–213. Addison-Wesley, 2001.

[PBvdL05] K. Pohl, G. Böckle, F. J. van der Linden. *Software Product Line Engineering : Foundations, Principles and Techniques.* Springer Verlag, 2005.

[Pet62] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Schriften des Instituts für instrumentelle Mathematik, University of Bonn, Germany, 1962.

[Poh97] K. Pohl. Requirements engineering. In A. Kent, J. Williams, C. Hall, editors, *Encyclopedia of Computer Science and Technology*, volume 36, pages 345–386. M. Dekker, New York, 1997.

[Pol03] P. R. Polsani. Use and abuse of reusable learning objects. *Journal of Digital Information*, 3(4):Article No. 164, 2003-02-19 2003.

[Por80] M. E. Porter. *Competitive Strategy.* Free Press, 1980.

[POS05] V. Pankratius, A. Oberweis, W. Stucky. Lernobjekte im E-Learning - Eine kritische Beurteilung zugrunde liegender Konzepte anhand eines Vergleichs mit komponentenbasierter Software-Entwicklung. In *9. Workshop Multimedia in Bildung und Wirtschaft*, Ilmenau, Germany, September 2005. Technische Universität Ilmenau.

[Pou96] J. S. Poulin. *Measuring Software Reuse: Principles, Practices, and Economic Models.* Addison-Wesley, 1996.

[Pre05] R. S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, international edition, 2005.

[Pro06] Produkthaftungsgesetz. http://beck-gross.digibib.net, November 2006.

[PS05a] V. Pankratius, W. Stucky. A formal foundation for workflow composition, workflow view definition, and workflow normalization based on petri nets. In S. Hartmann, M. Stumptner, editors, *Second Asia-Pacific Conference on Conceptual Modelling (APCCM2005)*, volume 43 of *CRPIT*, pages 79–88, Newcastle, Australia, 2005. ACS.

[PS05b] V. Pankratius, W. Stucky. Information systems development at the virtual global university: an experience report. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 639–640, New York, NY, USA, 2005. ACM Press.

[PS06] V. Pankratius, W. Stucky. A strategy for content reusability with product lines derived from experience in online education. In *Software Engineering Education in the Modern Age: Challenges and Possibilities*, volume 4309 of *Lecture Notes in Computer Science*, pages 128–146. Springer Verlag, 2006.

[PSS04] V. Pankratius, O. Sandel, W. Stucky. Retrieving content with agents in web service e-learning systems. In *The Symposium on Professional Practice in AI, IFIP WG12.6 – First IFIP Conference on Artificial Intelligence Applications and Innovations (AIAI)*, Toulouse, France, August 2004.

[PSV05] V. Pankratius, W. Stucky, G. Vossen. Aspect-oriented reengineering of e-learning courseware. *The Learning Organization: An International Journal*, 12(5):457–470, 2005. Emerald Group Publishing.

[Pus02] M. Pussinen. A survey on software product-line evolution. Technical Report 32, Institute of Software Systems, Tampere University of Technology, 2002.

[PV03] V. Pankratius, G. Vossen. Towards e-learning grids: Using grid computing in electronic learning. In *IEEE Workshop on Knowledge Grid and Grid Intelligence (in conjunction with 2003 IEEE/WIC International Conference on Web Intelligence)*, pages 4–15, Halifax, Nova Scotia, Canada, October 2003. Saint Mary's University.

[PV05] V. Pankratius, G. Vossen. Reengineering of educational material: A systematic approach. *International Journal of Knowledge and Learning (IJKL)*, 1(3):229–248, 2005.

[PW98] T. A. Phelps, R. Wilensky. Multivalent documents: A new model for digital documents. Technical Report CSD-98-999, Division of Computer Science, University of California at Berkeley, Berkeley, CA, USA, 1998.

[Rom05] D. Rombach. Integrated software process and product lines. In *International Software Process Workshop (SPW 2005)*, volume 3840 of *Lecture Notes in Computer Science*, pages 83–90, Beijing, China, May 2005. Springer Verlag.

[Roy70] W. W. Royce. Managing the development of large software systems. In *Proc. IEEE WESTCON*, pages 1–9, Los Angeles, CA, August 1970. Reprinted in Proc. of the Ninth International Conference on Software Engineering, March 1987, pp. 328-338.

[RRKP06] A. Reuys, S. Reis, E. Kamsties, K. Pohl. The ScenTED method for testing software product lines. In Käkölä and Duenas [KD06], pages 479–520.

[RU98] D. Robertson, K. Ulrich. Planning for product platforms. *Sloan Management Review*, 39(4):19–31, 1998.

[Sam90] P. Samuelson. Reverse-engineering someone else's software: is it legal? *IEEE Software*, 7(1):90–96, January 1990.

[SBF96] S. Sparks, K. Benner, C. Faris. Managing object oriented framework reuse. *Computer*, 29(9):52–61, 1996.

[Sch03a] M. C. Schlembach. *Information Practice in Science and Technology: Evolving Challenges and New Directions*. Haworth Information Press, 2003.

[Sch03b] K. Schmid. *Planning Software Reuse - A disciplined Scoping Approach for Software Product Lines*. PhD thesis, University of Kaiserslautern, 2003. Fraunhofer IRB-Verlag.

[Sch04] K. Schmid. Scoping als Basis optimierter Wiederverwendung. In Böckle et al. [BKPS04], pages 43–53.

[Scu03] A. Scupola. Organization, strategy and business value of electronic commerce: the importance of complementaries. In *In: J. Mariga (Ed.), Managing E-Commerce and Mobile Computing Technologies*. IRM Press, 2003.

[SG96] M. Shaw, D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[SG00] K. Schmid, C. Gacek. Implementation issues in product line scoping. In W. B. Frakes, editor, *6th International Conference on Software Reuse*, volume 1844 of *Lecture Notes in Computer Science*, pages 170–189. Springer Verlag, 2000.

[SGM02] C. Szyperski, D. Gruntz, S. Murer. *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.

[SHT04] H. M. Sneed, M. Hasitschka, M.-T. Teichmann. *Software-Produktmanagement*. Dpunkt Verlag, 2004.

[Sie04] G. Siemens. Learning management systems: The wrong place to start learning. http://www.elearnspace.org/Articles/lms.htm, November 2004.

[Sim95] M. A. Simos. Organization domain modeling (ODM): formalizing the core domain modeling life cycle. In *Proceedings of the 1995 Symposium on Software reusability (SSR '95)*, pages 196–205, New York, NY, USA, 1995. ACM Press.

[SLS02] C. Schlueter-Langdon, M. J. Shaw. Emergent patterns of integration in electronic channel systems. *Commun. ACM*, 45(12):50–55, 2002.

[SME07] SMETE. Science, mathematics, engineering and technology education (SMETE) digital library. http://www.smete.org/smete, January 2007.

[Som04a] D. Sommer. *Qualitätsinformationssysteme für E-Learning-Anwendungen*. PhD thesis, University of Karlsruhe, 2004.

[Som04b] I. Sommerville. *Software Engineering*. International Computer Science Series. Addison-Wesley, 7th edition, 2004.

[SS83] G. Schlageter, W. Stucky. *Datenbanksysteme: Konzepte und Modelle*. Teubner, 1983. 2nd edition.

[SS97] C. Schlueter, M. Shaw. A strategic framework for developing electronic commerce. *IEEE Internet Computing*, 1(6):20–28, 1997.

[SS03] S. Sánchez, M.-A. Sicilia. Expressing preconditions in learning object contracts. In *Proceedings of the Second International Conference on Multimedia and Information & Communication Technologies in Education (m-ICTE2003)*, Badajoz, Spain, December 3-6 2003.

[ST05] B. Simpson, F. Toussi. HSQLDB user guide. http://hsqldb.org, July 25 2005.

[Sta90] P. H. Starke. *Analyse von Petri-Netz-Modellen*. Teubner Verlag, 1990.

[STBZ$^+$05] K.-D. Schewe, B. Thalheim, A. Binemann-Zdanowicz, R. Kaschek, T. Kuss, B. Tschiedel. A conceptual view of web-based e-learning systems. *Education and Information Technologies*, 10(1 - 2):83–110, January 2005.

[Sun07] Sun. Sun developer network. http://java.sun.com, January 2007.

[SV99] C. Shapiro, H. R. Varian. *Information rules: a strategic guide to the network economy.* Harvard Business School Press, 1999.

[SV02] K. Schmid, M. Verlage. The economic impact of product line adoption and evolution. *IEEE Software*, 19(4):50–57, 2002.

[SvGB05] M. Svahnberg, J. van Gurp, J. Bosch. A taxonomy of variability realization techniques. *Software: Practice and Experience*, 35(8):705–754, April 2005.

[SW05] J. Sandrock, C. Weinhardt. System dynamics business models for e-learning content providers. In *The 2005 International Conference of the System Dynamics Society*, Boston, 2005.

[TDN03] S. Ternier, E. Duval, F. Neven. Using a P2P architecture to provide interoperability between learning objects. In *Proceedings of ED-MEDIA 2003 World Conference on Educational Multimedia, Hypermedia, and Telecommunications*, pages 148–151. AACE, 2003.

[TGTG05] P. Tessier, S. Gérard, F. Terrier, J.-M. Geib. Using variation propagation for model-driven management of a system family. In *Software Product Lines: 9th International Conference (SPLC 2005)*, volume 3714 of *Lecture Notes in Computer Science*, pages 222–233. Springer Verlag, 2005.

[The07] The Apache Software Foundation. The Apache ANT project. http://ant.apache.org, January 2007.

[Tic92] W. F. Tichy. Programming-in-the-large: past, present, and future. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 362–367, New York, NY, USA, 1992. ACM Press.

[TL05] L. A. Tedd, A. Large. *Digital Libraries: Principles and Practice in a Global Environment.* K.G. Saur, München, 2005.

[TMC99] S. A. Thibault, R. Marlet, C. Consel. Domain-specific languages: from design to implementation application to video device drivers generation. *IEEE Transactions on Software Engineering*, 25(3):363–377, 1999.

[TR03] A. Trifonova, M. Ronchetti. A general architecture for m-learning. In *Proceedings of the II International Conference on Multimedia and Information & Communication Technologies in Education (m-ICTE2003)*, Badajoz, Spain, December 3-6 2003.

[TR05] L. Thomas, E. Ras. Courseware development using a single source approach. In P. Kommers, G. Richards, editors, *World Conference on Educational Multimedia, Hypermedia and Telecommunications 2005*, pages 4502–4509, Montreal, Canada, 2005. AACE.

[TS04] S.-O. Tergan, P. Schenkel. *Was macht E-Learning erfolgreich?* Springer Verlag, 2004.

[TTC95] R. N. Taylor, W. Tracz, L. Coglianese. Software development using domain-specific software architectures: Cdrl a011 – a curriculum module in the sei style. *SIGSOFT Software Engineering Notes*, 20(5):27–38, 1995.

[TTK04] A. Tevanlinna, J. Taina, R. Kauppinen. Product family testing: a survey. *SIGSOFT Software Engineering Notes*, 29(2):12–12, 2004.

[Tur00] L. Turner. Automating Microsoft Office 97 and Microsoft Office 2000. *Microsoft Office 2000 Technical Articles*, March 2000.

[UK 07] UK LOM Core. UK Learning Object Metadata Core (UK LOM Core). http://www.cetis.ac.uk/profiles/uklomcore, January 2007.

[Uni07] University of Notre Dame. Latin dictionary and grammar aid. http://archives.nd.edu/latgramm.htm, January 2007.

[Var97] H. R. Varian. Versioning information goods. Technical report, University of California, Berkeley, March 13 1997.

[Var00] H. R. Varian. Markets for information goods. In *Monetary Policy in a World of Knowlege-Based Growth, Quality Change, and Uncertain Measurement*, 2000.

[Var05] H. R. Varian. Universal access to information. *Commun. ACM*, 48(10):65–66, 2005.

[VBDT04] P. Vidal, J. Broisin, E. Duval, S. Ternier. Learning objects interoperability: the ARIADNE experience. In *IFIP Congress Topical Sessions*, pages 551–556. Kluwer, 2004.

[VD04] K. Verbert, E. Duval. Towards a global component architecture for learning objects: A comparative analysis of learning object content models. In *ED-MEDIA 2004 World Conference on Educational Multimedia, Hypermedia and Telecommunications*, Lugano, Switzerland, 2004.

[vDdJK02] A. van Deursen, M. de Jonge, T. Kuipers. Feature-based product line instantiation using source-level packages. In *Software Product Lines: Second International Conference (SPLC 2)*, volume 2379 of *Lecture Notes in Computer Science*, pages 217–234. Springer Verlag, January 2002.

[vDKV00] A. van Deursen, P. Klint, J. Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[vdL02] F. van der Linden. Software product families in Europe: the Esaps & Café projects. *IEEE Software*, 19(4):41–49, 2002.

[vdML04] T. von der Maßen, H. Lichter. Deficiencies in feature models. In *Workshop on Software Variability Management for Product Derivation*, Boston, MA, August 2004. Helsinki University of Technology.

[Ver40] W. Vershofen. *Handbuch der Verbrauchsforschung (erster Band)*. Carl Heymanns Verlag, Berlin, 1940.

[vGBS01] J. van Gurp, J. Bosch, M. Svahnberg. On the notion of variability in software product lines. In *Proceedings IEEE/IFIP Working Conference on Software Architecture*, pages 45–54, 2001.

[VK00] T. Vehkomaki, K. Kansala. A comparison of software product family process frameworks. In *International Workshop on Software Architectures for Product Families (IW-SAPF-3)*, volume 1951 of *Lecture Notes in Computer Science*, pages 135–145. Springer Verlag, 2000.

[VW99] G. Vossen, M. Weske. The WASA2 object-oriented workflow management system. In *Proceedings of the 1999 ACM SIGMOD international conference on management of data*, pages 587–589, New York, NY, USA, 1999. ACM Press.

[W3C07a] W3C. HyperText Markup Language (HTML) Home Page. http://www.w3.org/MarkUp, January 2007.

[W3C07b] W3C. Resource Description Framework (RDF). http://www.w3.org/RDF, January 2007.

[W3C07c] W3C. Web services activity statement. http://www.w3.org/2002/ws/Activity, January 2007.

[War03] J. Ward. A quantitative analysis of unqualified Dublin Core metadata element set usage within data providers registered with the Open Archive Initiative. In *Proceedings of the 2003 Joint Conference on Digital Libraries*, pages 315–317, Houston, May 27–31 2003. IEEE Computer Society.

[WD05] G. Wöhe, U. Döring. *Einführung in die allgemeine Betriebswirtschaftslehre*. Vahlen, 2005.

[Web95] *Webster's College Dictionary*. Random House, Inc., 1995.

[Wil02] D. A. Wiley, editor. *The Instructional Use of Learning Objects*. Online version: http://www.reusability.org/read, February 2002.

[Wit96] J. Withey. Investment analysis of software assets for product lines. Technical report, Software Engineering Institute (SEI), Carnegie Mellon University, 1996. Technical Report CMU/SEI-96-TR-010.

[WK96] D. Weiss, H. Krcmar. Workflow-Management: Herkunft und Klassifikation. *Wirtschaftsinformatik*, 38(5):503–513, 1996.

[WL99] D. M. Weiss, C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, Boston, 1999.

[WLLS98] R. Y. Wang, Y. W. Lee, L. L.Pipino, D. M. Strong. Manage your information as a product. *Sloan Management Review*, 39(4):95–105, Summer 1998.

[WV01] G. Weikum, G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control*. Morgan Kaufmann, 2001.

[Yin03] R. K. Yin. *Case Study Research. Design and Methods*. Sage Publications, third edition, 2003.

[Zan02] C. Zanger. Leistungskern. In *Handbuch Produktmanagement*. Gabler, 2002.

[ZC03] D. Zubrow, G. Chastek. Measures for software product lines. Technical Report CMU/SEI-2003-TN-031, Software Engineering Institute (SEI), Carnegie Mellon University, 2003.

[ZMM06] C. Zannier, G. Melnik, F. Maurer. On the success of empirical studies in the international conference on software engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 341–350, New York, NY, USA, 2006. ACM Press.

[Zus94] H. Zuse. Complexity metrics/analysis. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 131–165. Wiley, 1994.

Index

Product Lines for Digital Information Products

The growth of the Web has spurred the creation and exchange of products which exist in digital form only, and which are created, distributed, and used exclusively in digital form. Digital information products are an important class of widely used digital products, whose core benefit lies in the delivery of information or education (e.g., electronic books, online newspapers, e-learning courses). This book introduces a novel and systematic approach, Product Lines for Digital Information Products (PLANT), which focuses on the creation of variants of such products within a product line, and which extends concepts from the area of software product lines.

This book and other material are available in electronic form at:
http://www.product-lines-for-digital-information-products.com

Questions or comments are welcome. You can send them to
info@product-lines-for-digital-information-products.com

ISBN: 978-3-86644-133-0

www.uvka.de