Software Product Lines:
State of the art

Patrick HEYMANS and Jean-Christophe TRIGAUX

Project: Product Line ENgineering of food TraceabilitY software
Financing: "Région wallonne" and "Fonds Social Européen"

First Europe Objectif 3

EPH3310300R0462 / 215315

15 September 2003
Abstract

Product lines engineering was born in the 80’s as an economic theory to increase economy of scale. Nowadays, the software product lines approach is promising, encouraging researchers and industrials to collaborate in order to increase software quality and to reduce software development costs. Convincing results have been achieved in specific domains, focusing on constructive (not opportunistic) reusability during software development. The purpose of this state of the art is to give a global overview on software product lines theories, presenting the general principles, some researches, methodologies and concrete applications. We insist on the small and medium companies aspects, underlining their difficulties to adapt this new approach to their specific needs and constraints.

Keywords: Software product lines, product families, Plenty, small and medium companies, software engineering methodology.
Contents

1 Introduction 9

2 PLENTY Project 10

I Software product lines paradigm 13

3 Software product lines general principles 13
   3.1 Domain Engineering ............................... 15
   3.1.1 Product line scope ............................. 15
   3.1.2 Core assets .................................... 15
   3.1.3 Production plans ................................ 15
   3.2 Application Engineering ........................... 16

4 Software product lines advantages 16

5 Software product lines disadvantages 17

6 Software product lines and Small Companies 19
   6.1 Problems ........................................... 19
   6.2 Proposed solutions ................................ 20

II Software product lines engineering 21

7 Variability 21
   7.1 Definitions of Variability ........................... 21
   7.2 Categories of Variability ........................... 22
   7.3 Levels of Variability ................................ 23
   7.4 Mechanisms of Variability ........................... 24
   7.5 Mechanisms & Levels of Variability ................. 24

8 Requirements 25
   8.1 Product Line Requirements engineering .............. 26
   8.2 Modeling Variability in requirements ............... 28

9 Product Line Architecture 29
   9.1 Product Line Architecture Definition ................. 29
   9.2 Product Line Architecture Evaluation ................ 29
   9.3 Product Line Architecture Evolution ................ 31
   9.4 Product Line Architecture Principles ................. 32
   9.5 Product Line Architecture Guidelines ................ 32
   9.6 Product Line Architecture Techniques ................. 33
      9.6.1 Architecture Recovery .......................... 33
      9.6.2 Architectural Design Model ...................... 34
      9.6.3 Product Line Architecture Description languages .. 34
   9.7 Modeling variability in product line architectures .... 35
      9.7.1 Product Line Architecture Variability and UML .... 36
      9.7.2 Product Line Architecture Variability and Discriminants 37
### CONTENTS

10 Components development ................................. 38
   10.1 Mining existing assets .............................. 39
   10.2 Commercial Off-The-Shelf ........................... 39

11 Traceability .............................................. 40

III Software product lines researches and applications .......... 43

12 Software product lines projects .......................... 43
   12.1 ARES .................................................. 43
      12.1.1 Objectives ......................................... 43
      12.1.2 Results ........................................... 44
   12.2 PRAISE ................................................. 44
      12.2.1 Objectives ......................................... 44
      12.2.2 Results ........................................... 44
   12.3 ESAPS .................................................. 44
      12.3.1 Objectives ......................................... 44
      12.3.2 Results ........................................... 44
   12.4 CAFÉ ................................................... 44
      12.4.1 Objectives ......................................... 44
      12.4.2 Results ........................................... 44

13 Software product lines development methods .................. 45
   13.1 FAST ................................................. 45
   13.2 FODA .................................................. 45
   13.3 FORM .................................................. 46
   13.4 RSEB ................................................... 46
   13.5 FeatuRSEB ............................................. 47
   13.6 ConIPF ............................................... 47
   13.7 PuLSE .................................................. 48
      13.7.1 Deployment Phases .................................. 49
      13.7.2 Technical Components ............................... 50
      13.7.3 Support Components ................................ 58
   13.8 KobrA ................................................. 59
      13.8.1 Framework engineering .............................. 60
      13.8.2 Application engineering ............................ 60
   13.9 Carnegie Mellon University Framework ................... 61

14 Software product lines applications ........................ 62
   14.1 Car Periphery Supervision (CPS) ........................ 62
   14.2 Celsius Tech: ship system 2000 .......................... 63
   14.3 Nokia: Mobile phones ................................... 63
   14.4 Weather Stations ...................................... 63
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Software product lines process [Nor02]</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>Software product lines risks [Coh02]</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>Software product lines development [Nor02]</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>Categorization of product family variability</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>Essential product family variability [HP03]</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>Technical product family variability [HP03]</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>Koala Component Model [vO02]</td>
<td>35</td>
</tr>
<tr>
<td>8</td>
<td>Order process system [Cla01]</td>
<td>36</td>
</tr>
<tr>
<td>9</td>
<td>Single discriminant [KM99]</td>
<td>37</td>
</tr>
<tr>
<td>10</td>
<td>Multiple discriminant [KM99]</td>
<td>38</td>
</tr>
<tr>
<td>11</td>
<td>Optional discriminant [KM99]</td>
<td>38</td>
</tr>
<tr>
<td>12</td>
<td>PuLSE Architecture [BFK+99]</td>
<td>49</td>
</tr>
<tr>
<td>13</td>
<td>PuLSE-BC process [BFK+99]</td>
<td>51</td>
</tr>
<tr>
<td>14</td>
<td>PuLSE-ECO process [BFK+99]</td>
<td>52</td>
</tr>
<tr>
<td>15</td>
<td>PuLSE-CDA process [BFK+99]</td>
<td>54</td>
</tr>
<tr>
<td>16</td>
<td>PuLSE-DSSA process [BFK+99]</td>
<td>55</td>
</tr>
<tr>
<td>17</td>
<td>PuLSE-I process [BFK+99]</td>
<td>56</td>
</tr>
<tr>
<td>18</td>
<td>PuLSE-EM process [BFK+99]</td>
<td>57</td>
</tr>
<tr>
<td>19</td>
<td>Car Periphery Supervision systems [TFF+01]</td>
<td>62</td>
</tr>
</tbody>
</table>
1 Introduction

During the software development process, various problems can occur. Firstly, in specific domains, the actual development of software is mainly seen as redevelopment. Most products have been built before at some levels. Industries, such as telecommunications, home appliances or car industry, build the same or similar products over and over again.

Secondly, the software producer’s interests are in contradiction with the customer’s interests. Indeed, producers want to maximize their benefits and, thus, to minimize their production’s costs and time to market. In opposition, customers ask for better quality and for software tailored to their individual needs.

Thirdly, complexity and size of software products are rapidly increasing due to the market’s evolution. Moreover, the higher the products’ diversity is, the bigger the complexity and challenges for an organization are.

The following list summarizes typical problems which can arise as complexity of products increases:

- The same functionality is developed at different places.
- The same changes are repeated at different places.
- Identical features behave differently depending on the particular product.
- Updating products and managing changes are challenging tasks.
- Maintaining products is asking too much efforts and resources.

The software product lines approach proposes solutions to cope with those problems. Hence, there is a need for organizations to learn how to manage a product line or how to improve their way of managing it.

Following the usual programming methods based on subroutines, modules, objects and component based systems, software product lines seem to become a new step forward in the field of components reusability. Software product lines represent an innovative and growing concept in software engineering. This approach is based on a development process including both development for reuse and development with reuse. Software product lines offer a new strategy, for the programmers, to cope with the emergence of new similar products. Indeed, a new product is not really implemented but more integrated in a product line, adding new necessary components, reusing common assets, using preplanned variation mechanisms such as parameterization or inheritance.

Nevertheless, the companies who can afford software product lines engineering, are mainly large ones, who can support important initial investment and wait for a long term return on investment. On white papers, software product lines’ theories are really interesting and promising. But this is not the perfect solution, as their applicability in a concrete domain is not so easy, especially for small companies. That is the reason why, there is many interests to revise those techniques for small companies, who have
short term deadlines and must stay flexible to respond to customer’s requests and market pressure. This adaptation is a central goal directing the PLENTY project.

This state of the art is a deliverable of this project and proceeds as follows. First, we present shortly the PLENTY project and, secondly, we describe three main parts.

The first part introduces the software product lines paradigm, presenting the general principles, their advantages, disadvantages and the small and medium companies aspects.

The second part introduces the software product lines engineering, presenting various software development concepts adapted to the software product line’s context. Thus, variability, requirements, product line architecture, component development and traceability are described.

The third part introduces some software product lines researches and applications, presenting various software product lines research projects, development methodologies and concrete applications.

2 PLENTY Project

PLENTY stands for ”Product Line ENgineering of food TraceabilitY software”. It is a ”FIRST EUROPE OBJECTIF 3” project taking place at the University of Namur, in the Software Requirements Engineering Laboratory, under the supervision of Prof. Patrick Heymans and Prof. Pierre-Yves Schobbens. This project, funded by the "Ministère de la Région wallone - DGTRE" and the "Fonds Social Européen", started on March, 1 2003 and will end on February, 28 2005. The project aims to develop, apply and diffuse a methodology based on software product lines to increase reusability in the all software development life-cycle. This methodology is particular, in the sense, that it covers various aspects like suitability for small companies, facilitating certification of safety-critical products, covering the whole range of software engineering artifacts, from requirements to code, and guaranteeing conformance of the latter with the former.

This project is realized in collaboration with the Software System Engineering laboratory (SSE) directed by Prof. Klaus Pohl from the University of Essen (Germany) and with BIVTrace, a small company dealing with food traceability software from Fal-isolle (Belgium).
The objective is to define and reuse a bone of reusable software components, which are directly linked with the commonalities of the product line. In practice, the main idea is to propose a method, based on requirements engineering and software product lines, that helps small companies with various tasks like:

- Define a list of products that should be contained in the software product line,
- Identify and model commonalities and variabilities,
- Define a software product line architecture,
- Define a library of reusable components corresponding to commonalities,
- Define a library of components corresponding to variabilities,
- Define how to reuse this architecture and those libraries in order to develop new products.
Part I
Software product lines paradigm

3 Software product lines general principles

“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”
[CN03].

Product line engineering is a process that delivers reusable components, which can
be reused to develop a new application for the domain. Indeed, a software product
line defines, software product lines requirements, a software architecture and a set of
reusable components, shared by the products, which implements a considerable part of
the products’ functionalities.

The purpose is to reduce the time and costs of production and to increase the soft-
ware quality by reusing elements (core assets) which have been already tested and
secured. These objectives can be realized by putting in common development arte-
facts such as requirement’s documents, conception’s diagrams, architectures, codes
(reusable components), test’s procedures, maintenance’s procedures, ...

The general process of product lines is based on the reusability of requirements,
arithmetic and components. As shown in figure 1, there are two main phases. The
first one is called "Domain Engineering", or development for reuse, and consists in
developing core assets through the domain analysis, domain design and domain imple-
mentation processus. The second one is called "Application Engineering", or devel-
opment with reuse, and consists in developing the final products, using the core assets
and the specific requirements expressed by the customers. This phase is composed of
three processes: application requirements, application design and application coding.
In details, the study of the domain gives a set of reference requirements which can be reused to define the requirements of an application and to explicit the necessity to integrate new requirements. A reference architecture, defined by the domain design, is used to develop and structure applications. Reusable components are reused to code applications.

A backward and forward traceability must be established between the reference requirements, the reference architecture and the reusable components to facilitate the changes and updates management in the product line. This notion is crucial for software product lines, as its aim is to assure quality and to support change integration.

During the application engineering phase, a feedback process can be used to revise the domain design and the domain implementation. Indeed, new products may reveal the necessity to integrate new reusable components to the product line’s architecture or to modify reusable components.

The mechanism behind the product line theory [Nor02] is divided in three principal processes which are not sequential but more parallel: the core assets development, the product development and the management. In fact, products and core assets are influenced by each other and evolve in parallel during their life cycle. For instance, core assets are the base used to produce new products, but they are also defined thanks to preexisting products and future products, and they evolve as new products are developed.
3.1 Domain Engineering

The Domain Engineering is in charge with the core assets development. This phase is called the development processus for reuse and classically follows a waterfall life cycle. If we speak in terms of processus, the core assets development uses various inputs like product constraints, production constraints and production strategy. This process wants also to reuse inventory of pre-existing components which could have been maintained before. The development experiences of the company are also reused, through the previous styles, patterns and frameworks. The domain engineering gives in outputs: the product line scope, the core assets and the production plan which are described in the following.

3.1.1 Product line scope

This scope is simply represented by a set of products that the product line can integrate. Software product line scope is a description of the products that will constitute the product line. This scope may consist of an enumerated list of product names. More typically, this description is cast in terms of the characteristics that the products have all in common, and the ways in which they vary from one another [CN03]. It’s a statement, made by the company, about what systems will be build as part of the product line and what systems will not.

This set will obviously vary with the time, with the various organization’s strategies and with the markets’ evolution. The definition of the scope is a crucial point, because too broad a scope renders the core assets too complex to be effectively reused and too narrow a scope does not justify the cost of core assets development and maintenance.

3.1.2 Core assets

Core assets are those assets that form the basis for the software product lines. Core assets often include, but are not limited to, reference requirements, reference architecture, reusable software components. More documents can be also included, such as domain models, documentation and specifications, performance models, schedules, budgets estimation, test plans, test cases, work plans, process descriptions,... [CN03].

3.1.3 Production plans

The production plans are used to specify a determined way to derive a product from the core assets. Thus a production plan defines how the core assets will be used to build a product, it simply corresponds to a developers’ guide which helps them to correctly and efficiently use core assets.

A production plan specifies the product line’s constraints that must be followed by the products. It defines and maintains the link between the core assets and the products.
3.2 Application Engineering

The application engineering is in charge with the products development. This second phase is called the development process with reuse and follows a classical waterfall life cycle, reusing the core assets already defined in the previous step. The development of final products is based on the core assets and on the specific customers’ requirements.

If we speak in terms of processus, the products development process uses as inputs: the outputs of the previous phase (product line scope, core assets, production plans) and a delta of requirements that expresses the variations. The application engineering generates the final specific products as outputs.

4 Software product lines advantages

Software product line’s theories can generate various benefits classified in three types: organizational benefits, software engineering benefits and business benefits.

**Organizational benefits** regroup advantages like a better domain’s comprehension, the facility to train people, the high quality product and the customer’s trust.

**Software engineering benefits** include advantages like reusability of requirements and their components, better analysis of requirements, another view on the requirements for the client, control of software quality, establishment of programming standards, a way to find and erase redundant implementations and complete reusable documentations.

And last but not least, **business benefits** concern the reduction of production, maintenance and test costs (thanks to the reuse of commonalities between various products). Moreover, product lines generate a better efficiency in the processus and the possibility to improve the budget and time planning. A detailed description of the benefits can be found in section “benefits and costs of product line” in [CN03].

In [BCK03], several concrete statistics are presented to illustrate the improvements in costs, time to market and productivity:

- Nokia is able to produce 25 to 30 different phone models per year (up from 4 per year) because of the product line approach.
- Cummins, Inc., was able to reduce the time it takes to produce the software for a diesel engine from about a year to about a week.
- Motorola observed a 400% productivity improvement in a family of one-way pagers.
- Hewlett-Packard reported a time to market reduced by a factor of seven and a productivity increase by a factor of six, in a family of printer systems.
5 Software product lines disadvantages

One of the main disadvantages of the software product lines approach is its application in a specific company’s domain. Indeed, it generates important investments, risks and impacts on the organization and on the management, in terms of mentality evolution and of time and money costs. That’s why management must focus on a middle term product line strategy and not on immediate results. In the next paragraphs, we describe some of the most challenging problems encountered.

Changing the organization’s original development mode from a single product view to a product line approach entails a fundamental shift for the organization and for the mentalities that can bring resistance to changes.

Another problem arises as only a few software engineers have a global view on the entire product line architecture. In the various case studies presented in Clements Book [CN01], he emphasizes the need of a software product line champion who knows perfectly the application domain, who has enough responsibility, authority and experience inside the company and who has enough motivation and understanding of software product lines’ theories.

The software product line creates also an important need in information distribution through the organization. Indeed, the architecture and the way it evolves must be known by every one in time. Nevertheless, in small companies, the communication is often assured by mutual adjustment which allows rapid exchange of information between those concerned.

Some difficulties can appear, when a decision must be taken to merge or not a new product with the product line. The domain context evolution requires to fix cautiously the product line scope. Indeed, as mentioned before, too broad a scope renders the core assets too complex to be effectively reused, too narrow a scope does not justify the cost of core assets development and maintenance.

Designing product line architectures is difficult, due to the lack of guidelines, techniques and tools to represent or validate reference architectures and integrated tools to develop and exploit software product lines.

Thus software product lines introduction is hindered by technical and organizational risks. In [Coh02], different problems (see figure 2) are listed based on a questionnaire underlining that the organizational and investment resistance are the most critical.

When we compare traditional and software product lines development (see figure 3), we can underline that the software product lines approach needs a long term management, because the visible benefits are not immediate. Indeed, the initialization of a software product line requires a considerable launching investment and the break even point would, generally, be reached at middle term, delaying the return on investment. A sufficient number of products must be developed in order to appreciate the real advantages.

In the context of small companies, we must refine this view as most of them can not assume important initial investments. Those companies are also put under pressure
Table: Software product lines risks

<table>
<thead>
<tr>
<th>Problems</th>
<th>Percent responding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organizational resistance</td>
<td>52 %</td>
</tr>
<tr>
<td>Management resistance</td>
<td>36%</td>
</tr>
<tr>
<td>Developer resistance</td>
<td>32%</td>
</tr>
<tr>
<td>Concerns over large investment</td>
<td>45%</td>
</tr>
<tr>
<td>Lack of properly trained staff</td>
<td>29%</td>
</tr>
<tr>
<td>Inability to measure impact</td>
<td>19%</td>
</tr>
<tr>
<td>Concerns of long lead time</td>
<td>18%</td>
</tr>
</tbody>
</table>

Figure 2: Software product lines risks [Coh02]

by the market and must cope with short term deadlines in order to stay competitive. Moreover, the risks encountered can not be precisely evaluated and supported. Thus, small companies’ managers are often reluctant to introduce those software product lines theories.

Figure 3: Software product lines development [Nor02]
6 Software product lines and Small Companies

One main objective of the PLENTY project is to adapt software product lines methodologies for small and medium companies, which can not support huge investment and risks, automatically generated by this kind of solution.

Indeed, several methodologies have been successfully used in important companies like Bosch, Nokia, Phillips, Siemens, ...

Nevertheless, in this type of firms, the software development process is directed by mass market, and standard products requirements are designed by market analysis and evolution. In the small and medium companies’ context, the situation is relatively different as the products developed are individual products which correspond to customers specific needs and requirements. This products classification, proposed in [HP03], emphasizes the distinction between individual and standard products which is essential to understand the small and medium companies’ context specificity in software product lines.

A software product line does not appear accidentally, but requires a conscious and explicit effort from the interested organization.

Firstly, the organization may take an **evolutionary or a revolutionary approach** for the adoption process. Thus, the company has the choice to create its software product line from scratch (using the old system at the same time to continue the production) or to evolve its actual system incrementally.

Secondly, the product line approach can be applied to an **existing line of products** or to a **new system**.

Each case has associated risks and benefits. For instance, in general, the revolutionary approach involves more risks, but higher returns compared to the evolutionary approach. Nevertheless, we can rarely expect a small or medium company to realize a complete product line life-cycle immediately from the start. Most would start small, one component at a time, within pilot projects.

Clearly, the adoption of such an approach implies specific difficulties that must not be neglected to prevent avoidable fails. In the following, we insist on a non-exhaustive list of specific problems associated to small and medium companies context. We also propose a list of suggestions to prevent these problems. The purpose of the next section is not to repeat the various general disadvantages of product lines, but to detail possible difficulties for small and medium companies.

### 6.1 Problems

1. **Difficulties to get support**, as the initial investment will generally delay one or more products in their time-to-market, which is often considered as a major problem, despite the future benefits. In fact, the costs estimation to develop core assets is hard to evaluate.

2. **Difficulties to be flexible**, because to be efficient, software product lines need to
have domain stability and a clear vision about the future evolution of the company’s applications.

3. **Difficulties to make evolve** the product line due to the dependencies existing between the core assets.

4. **Difficulties to find a champion.** A champion is a person which has enough experience, responsibility and authority inside the company, a person which has a general overview on the application domain and a person which is motivated and ready to implicate herself in the software product line approach. This kind of person is really difficult to find in small and medium companies because she does not exist or she does not have the time to take this task in charge.

5. **Difficulties to introduce new technologies**, if no development process has been previously explicitly defined and used in the company. Indeed, if no development process exists, introducing a new process can be considered, but with the pressure increasing in a project this process will be rapidly ignored and previous habits reestablished. [KMSW00]

6. **Difficulties to define a reference architecture**, as resistance to changes can occur. Indeed, architects want to avoid as much rework as possible and do not easily accept that their architecture can be improved. [KMSW00]

### 6.2 Proposed solutions

1. Apply the software product lines approach to a typical and well-understood sub-domain which is not critical to business success.

2. Convince key people about the benefits of product lines to avoid deadlock in technology transfer.

3. Collect and analyze the requirements of existing and future products in the product line and, based on that analysis, identify conflicts and variations between products.

4. Keep it simple and adapted to future needs

5. Provide detailed documentations with, first of all, requirements descriptions and an explicit global vision of the product line architecture and other core assets.

6. Describe an explicit process for the product line and core assets evolution.

7. Provide examples in the companies domain.

8. Create awareness about software product lines engineering principles which have impacts on development’s practices and on projects’ definitions.
Part II
Software product lines engineering

7 Variability

Variability shows how the product line allows for and facilitates the differences between the products contained in it. Many implementation approaches, used in industry for handling variability in a product line, suffer when new products are engaged in, or when existing products are evolving (scalability). This impacts, at first, the overview over the variation leading gradually to a degraded implementation structure.

Variability plays a central role in the entire software product lines development process. Indeed, variability concerns the all development life cycle from the requirements elicitation to the architecture, the components, the coding and the tests. Each variability encountered in a product line context can be connected with a corresponding feature varying under specific conditions.

A simple example of variability can be found in the telecommunication domain. Indeed, if we consider a mobile phone product line, variabilities are obvious to identify: the number of keys, the screen’s dimensions, the languages, the games,....

In this section, we first define and classify the notion of variability. Then, we describe the variability levels. In order to manage variability, we present various possible variability mechanisms and their applicability through the variability levels.

7.1 Definitions of Variability

Commonalities consist of a structured list of assumptions that are true for each member of the family. David M. Weiss and Chi Lai define variability in product family development as “An assumption about how members of a family may differ from each other”[WL99]. Variabilities specify the particularities of a system corresponding to the specific expectations of a client.

Variability is usually described with variation points and variants. A variation point locates a variability and its bindings by describing several variants. Each variant corresponds to one alternative designed to realize a particular variability and, thus, to bind it in a concrete way. In other words, the variation point locates the insertion point for the variants and determines the characteristics (attributes) of the variability.

Difficulties occur to manage variability in software product lines, because both single products and product family variants may change. Products evolve during the time with classical releases like in single product development. Furthermore, they can also evolve following their variants’ choice. This two variation aspects are called in [HP03] “variation in time and in space”. Variation in time concerns the different releases of a product. Variation in space means that at the same time two products may contain variants which have different implementation versions.
7.2 Categories of Variability

A classification of variabilities, shown in figure 4, has been proposed by Felix Bachmann and Len Bass [BB01] following several criteria:

- **Variability in function** means that a particular function may exist in some products and not in others.
- **Variability in data** means that a particular data structure may vary from one product to another.
- **Variability in control flow** means that a particular pattern of interaction may vary from one product to another.
- **Variability in technology** concerns the platform (OS, hardware, dependency on middleware, user interface, run-time system for programming language) which may vary in a similar way to function but with the technical point of view.
- **Variability in product quality goals** means that goals like security, performance or modifiability may vary from one product to another.
- **Variability in product environment** means that the product domain may impose specific requirements.

![Figure 4: Categorization of product family variability](image)

Another classification, based on the distinction between essential and technical variability, has been proposed by Günter Halmans and Klaus Pohl [HP03]. **Essential variability** corresponds to the customer viewpoint who cares about usage aspects concerning functional and non-functional requirements. **Technical variability** corresponds to the product family engineer viewpoint who cares about variation points, variants, binding times to realize variability. Clearly, essential variability defines what to implement and technical variability defines how to implement it. These two classes of variability have also been refined in various subcategories which are summarized in figures 5 and 6. For more details refer to [HP03].
7.3 Levels of Variability

Various levels of variability are defined by [SB00] in order to identify the level of abstraction.

- **Product Line Level.** Variability on this level is concerned with how products differ, i.e. which components are used by different products and which product specific code (PSC) is used.

- **Product Level.** The variability is concerned with the architecture and choice of components for a particular product. On the product level, the components are fitted together to form a product architecture, and the PSC is customized for the particular product variation. The variability issues on this level are:
  1. how to fit components together,
  2. how to cope with evolving interfaces, and
3. how to extract and/or replace parts of the PSC.

- **Component Level.** This level is where the set of framework implementations are selected. The variability issues on this level are:
  1. how to enable addition and usage of several component implementations
  2. how to design the component interface in such a way that it survives the addition of more concrete implementations.

- **Sub-Component Level.** A component consists of a number of feature sets. On the sub-component level the unnecessary features are removed to avoid dead code and others are added.

- **Code Level.** The code-level is where evolution and variability between products are implemented, ideally matching the provided and the required class interface defined in previous steps.

### 7.4 Mechanisms of Variability

Jacobson defines, in [JGJ97], various mechanisms to manage variability. In the following table, they are described with the type of variability they address and the development stage (requirements time, run time,...) when to apply them.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Time of Specialization</th>
<th>Type of Variability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td>At class definition time</td>
<td>Specialization is done by modifying or adding to existing definitions.</td>
</tr>
<tr>
<td>Extension</td>
<td>At requirements time</td>
<td>One use of a system can be defined by adding to the definition of another use.</td>
</tr>
<tr>
<td>Uses</td>
<td>At requirements time</td>
<td>One use of a system can be defined by including the functionality of another use.</td>
</tr>
<tr>
<td>Configuration</td>
<td>Previous to runtime</td>
<td>A separate resource, such as file, is used to specialize the component.</td>
</tr>
<tr>
<td>Parameters</td>
<td>At component implementation time</td>
<td>A functional definition is written in terms of unbound elements that are supplied when actual use is made of the definition.</td>
</tr>
<tr>
<td>Template instantiation</td>
<td>At component implementation time</td>
<td>A specification type is written in terms of unbound elements that are supplied when actual use is made of the specification.</td>
</tr>
<tr>
<td>Generation</td>
<td>Before or during runtime</td>
<td>A tool that produces definitions from user input.</td>
</tr>
</tbody>
</table>

### 7.5 Mechanisms & Levels of Variability

In [SB00], Mikael Svahnberg and Jan Bosch describe the variability mechanisms applicability in each variability level.

They underline the central role played by two variability mechanisms; the configuration and the parameters. Indeed, configuration is useful for the higher variability levels (i.e. the product line, product and component levels). Parameters are useful for the lower levels (i.e. the components, sub-components and code levels). Other mechanisms such as inheritance and extension are also used at all levels of variability because
they can be easily combined with other mechanisms.

Nevertheless, in industry, the parameterization is often used as the only mechanism for all variability levels using ifdefs, for instance. This bad habit implies many problems and difficulties to manage variability at higher levels and, thus, to maintain an efficient and coherent software product line.

8 Requirements

A requirement is defined as "A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document. The set of all requirements forms the basis for subsequent development of the system or system component" [Eas03].

The requirements capture the characteristics of a system. A software system has to satisfy all the requirements that are specified in the problem domain. There are two kinds of requirements: functional (concerning the behavior of the system) and non-functional (concerning performance, reliability, memory size,...). Nevertheless, this distinction is not always pertinent. Indeed, if we elicit requirements for some critical systems, the reliability could be described as a functional requirement.

Software product lines requirements define the products and their features in the product line. One of the most interesting idea, in the software product lines analysis, is the fact that a set of requirements can be identified as common in the earlier and future systems. Thus, the requirements can be classified in two main types; the first ones are called commonalities and express the common requirements determined for a group of systems, the second ones are called variabilities and specify the delta of requirements or the increment to the generic set needed to specify a particular system, corresponding to the specific expectations of a client.

The main advantage of this technique is the commonalities reusability that brings many opportunities in terms of simplifications for new systems specifications. Common requirements across the entire family clearly constitute a core asset. Moreover, as the definitions of commonalities can be reused, more time can be allocated to the definition of variabilities.

However, evolution in software product lines is more complex, compared to traditional software development. New requirements that must be incorporated into the product line may enter in conflict with requirements originating from already existing products. Furthermore, changes in the requirements make the software product line evolve over time. These changes can originate from customers using the products or from the introduction of new needs and new products.
8.1 Product Line Requirements engineering

Requirements engineering includes five important activities (elicitation, analysis, specification, verification, management) that must be reconsidered in the context of software product lines:

1. The requirements elicitation

Requirements elicitation is the process of discovering, reviewing, documenting, and understanding the user’s needs and constraints for the system [CN03].

Requirements elicitation for software product lines must have a global view on the application domain. Indeed, this elicitation concerns all the products contained in the product line scope. The idea is that, during the elicitation, we have to differentiate the common requirements and the specific requirements between all of these products.

One particularity of the elicitation for software product lines is the need to represent and anticipate the variations expected to occur during the lifetime of the product line. The idea is that this kind of elicitation implies a huge work to understand the domain and its future evolution, touching more stakeholders than in the single-system requirements elicitation. This first step, as in classical software development, is crucial, but the usual elicitation techniques are not adapted to clearly identify variations and must be revised.

2. The requirements analysis

Requirements analysis is the process of refining the users needs and constraints [CN03].

The requirements analysis for software product lines involves the clear identification of commonalities and variabilities. In this step, the possibility of reusability within the product line must be identified. The preplanned reusability expressed in terms of requirements can be extended to the concrete software components of a system, which must be linked to the commonalities. The advantage is that those components, implementing the commonalities, should not be reinvented and that all the code, documentation, data structures, algorithms, design decisions, maintenance procedures and tests can be reused.

3. The requirements specification

Requirements specification is the process of documenting the user’s needs and constraints clearly and precisely [CN03].

The requirements specification for software product lines must include the distinction presented above for the requirements. Thus, one set of commonalities and one set of variabilities or product specific requirements must be clearly specified. In this step, an important point should be considered; the analyst must pay
8.1 Product Line Requirements engineering

attention to the way the variations should be integrated to the commonalities. Various approaches can be considered, like simply adding new specifications for new functionalities, or using parameterization, or using extension mechanism or instantiating commonalities.

4. The requirements verification

Requirements verification is the process of ensuring that the system requirements are complete, correct, consistent, and clear [CN03].

First of all, the commonalities should be considered as highly critical for the entire software product line. This is the reason why requirements verification is really important and must be fully achieved. This additional work will pay off in middle term as this verification will be done for the entire product line. We must also consider the verification of variation requirements for a specific product, keeping in mind that they must stay coherent with the commonalities.

5. The requirements management

Requirements management is the process of scheduling, coordinating, and documenting the requirements engineering activities (i.e., elicitation, analysis, specification, and verification)[CN03].

The requirements management for software product lines must really pay attention to requirements traceability that links the requirements backwards to their sources (stakeholders) and forward to the resulting system development work products (components). Indeed, a formal or semi-formal mechanism should propose necessary modifications needed to respond to change management policies and impacts that this kind of policies will bring.

A new product, entering in a product line, is supposed to support more functionalities or the same functionalities in a different way. This is achieved by either adding a new component, or by changing an existing component. In some cases, the changes required are so vast that it is more cost-effective to simply rewrite the component from scratch. Improvement of functionalities makes the product line evolve with the client’s needs.
8.2 Modeling Variability in requirements

The increase of variability leads to a situation where the complexity of managing it becomes a primary concern during software development. To express variability, it is necessary to model variation points and their variable parts [JGJ97]. Variability must be treated already during the requirements engineering phase. Indeed, a key to success, in the initial product line project phase, is an efficient communication between the designers and the stakeholders about requirements and thus about variabilities.

Modelling notations can be graphical, textual or a mixture of both. Modelling notations can use different techniques like feature diagrams, use cases or class diagrams. We simply enumerate here some notations used to model variability in software product lines requirements.

Concerning feature diagrams notations, we can find in literature: FODA notation [KCH+90], FORM notation [KLD02], FeatuRSEB notation [GFd98], Bosch’s notation [vGBS01] and Riebisch’s notation [RBSP02].

Concerning use case models, we can find in literature: Gomaa’s notation [GS02], Halmans’ notation [HP03], Bertolino’s notation [BFG+02], John and Muthig’s notation [JM02] and van der Maßen’s notation [vdML02].

Concerning UML class diagrams, we can find in literature: Clauss’ notation [Cla01].
9 Product Line Architecture

In this section, first the product line architecture is defined. Then, the product line architecture evaluation and evolution are presented. The next point describes some general principles and guidelines concerning the product line architecture. Moreover, we introduce interesting techniques related to product line architecture, such as the architecture recovery, the architectural design model from Bosch [Bos00] and architecture description languages [vO02], [DvdHT02]. Finally, various notations designed to model variability in product line architecture are introduced.

9.1 Product Line Architecture Definition

The notion of software architecture is defined as "The fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution. [Gro99]"

The notion of product line architecture is defined as "a single specification capturing the overall architecture of a series of closely related products" [Muc03].

When the decision to initiate a software product line has been taken, the first step is the domain analysis describing the variability in the requirements, the second important step is the definition of the product line software architecture. This core asset must support the functional and quality requirements of the products included in the product line. The role of the software product line architecture is mainly double; first, it must describe the commonalities and variabilities of the products contained in the software product line and, secondly, it must provide a common overall structure. Product line architectures are designed for families of related applications and applications construction is accomplished by composing reusable components. Basically, a product line architecture is composed of a set of architectural decisions, a set of reusable components and eventually a library of optional components corresponding to specific clients' requirements. The various components can be assembled together following the implicit rules of the planned architecture.

9.2 Product Line Architecture Evaluation

It makes sense to evaluate the architecture to be sure that it will be robust to support the quality goals of the product line and the individual products that will be built from it.

The software product line architecture contains critical core assets which must be constantly evaluated to confirm their adequacy with the actual requirements. Moreover, the assessment can detect early, if the architecture is unnecessarily complex or if it ignores some important requirements. Various techniques may be used for this task: qualitative measures such as scenarios, checklists, questionnaires and quantitative measures such as metrics, simulations, prototypes, experiments. [Har01].

A scenario can be presented as a brief description of interactions between the stakeholder and the system. For instance, a stakeholder, concerned with modifiability, creates scenarios that propose a set of specific changes to be made to the system.
A **checklist** is a detailed set of questions, that represents the collected experiences from many evaluations of similar systems. As they are the result from evaluations of similar systems, they tend to be domain-specific and are often focused on particular qualities of the system.

A **questionnaire** is a list of general and relatively open questions that apply to all architectures. Questions can target the process that has generated the architecture, the architecture description, as well as how the architecture fulfills various quality requirements.

A **metric** is a quantitative interpretation of a measurable entity on the architecture, for instance, the number of components each component depends on.

A **prototype** is a model of the architecture and in most cases it covers the parts of the architecture raising the most relevant design questions.

A **simulation** requires a model that reflects relevant parts of the architecture. The quality attributes to assess will determine the necessary level of details in the model and the analytic techniques to use. It exists several analytic techniques for assessing various qualities in software architectures:

- Rate-Monotonic Analysis (RMA) to estimate worst-case response time;
- Queuing models to predict performance;
- Petri nets to argue about safety properties;
- Markov models to reason about reliability.

A prototype or a simulation of the system can provide information that clarifies important architectural issues and, therefore, helps in creating the architecture. The main advantage of a prototype or a simulation is that their behavior relates to actual use of the system. The main disadvantage is the cost to obtain the prototype or the models used for simulation.

Moreover, various architecture evaluation techniques exist in the literature. For instance, the Architecture Trade-off Analysis Method (ATAM) is “a scenario based architecture method that focuses on a system’s quality goals” [CKK01].
9.3 Product Line Architecture Evolution

When a new set of products should be developed, an architecture evaluation must be done and various decisions could be taken to integrate it. A first solution is to introduce the new requirements directly in the components themselves. Indeed, it is possible to add new framework implementations to the components or to change framework implementations or to increase/decrease the functionalities of the components. On another level, it is possible to concentrate the effort on the architecture itself:

1. Split in product line architecture

   A possibility is to split the software product line into two, using the existing product line architecture as a template for creating another. The product families, and hence the product line architecture, evolve into various directions, but may keep similar traits.

2. Inheritance in product line architecture

   A second approach is to use the mechanism of inheritance to define a sub product line. This way, functionalities which are common for the complete product line are still shared, and other products can directly benefit from bug fixes and other improvements.

3. Extend in product line architecture

   A third approach is to extend the product line architecture with new components corresponding to specific requirements which can not be easily adapted to the current product line components. This method implies that the relations between the new components and the old ones must be taken into account.

4. Change in product line component

   The adoption of new requirements may introduce modifications in components. Those modifications must not be directly integrated in the components but the entire component must be rewritten from scratch to avoid risks. This leads to replace the old component by a new one.

5. Relation between components

   As the components in the architecture are linked with each other, the introduction of new components requires new relations between components contained in the product line architecture. Indeed, new requirements may require one component to request services from another. However, the relations between various components may simply change.
9.4 Product Line Architecture Principles

In [DKOW97], Dikel defines six basic architecture principles in order to guide software product line engineers through architecture creation:

1. Focusing on simplification, minimization, and clarification
2. Adapting the architecture to future customers’ needs, technology, competition, and business goals
3. Establishing a consistent architectural rhythm, generating regular architecture and product releases that help to coordinate the actions and expectations of all parties.
4. Partnering and broadening relations with stakeholders.
5. Maintaining a clear architecture vision across the company.
6. Pro-actively managing risks and opportunities.

9.5 Product Line Architecture Guidelines

In [SB99], Mikael Svahnberg and Jan Bosch have defined some guidelines to help software product line engineers to elaborate and maintain a reusable architecture:

1. Generalize component interfaces

When, first, designing the architecture, the interfaces definitions must be general enough to cope easily with the addition of new implementations in components. Thus, defining interfaces implies to be proactive as future products requirements must be handled too.

2. Avoid adjusting component interfaces

The main idea is to delay as much as possible the component interfaces growing or shrinking, due to the addition or the suppression of functionalities. Growing the interface results mainly from external components which need more functionalities. Shrinking interface is more critical, as it results mainly from the introduction of a new component which specialized a functionality previously handled in another component. Thus, the impact on the implementation is huge as many other components may use the previous functionality. So, the component itself must be changed, the new one must be correctly implemented, the various components using the previous functionality must be identified and modified.

3. Separate domain and application behavior

The distinction between domain and specific applications functionalities must be kept during the implementation. The aim is to prevent mixing this two types of functionalities which evolve differently. This allows to treat them separately using aggregated relationships, as the only connector between them, and to increase the domain functionalities reusability.
4. Avoid split software product lines

The unity of a software product line is really important. A product line is usually defined for a domain but not for a specific type of products. So, during the product line evolution, splitting the product line according to product types must be banished. Indeed, in the food traceability software context, the idea is to create one software product line reusable for many product types like salmon, meat, cheese, syrup, roll,... Furthermore, if various versions of the product line appeared, the different versions should be joined. Indeed, maintaining two product lines in parallel is quite expensive and inefficient.

5. Detect and exploit common functionalities in component implementations

A library of implemented common functionalities shared between implementations must be managed for each domain.

6. Be open to rewrite components and implementations

Situations exist where a particular implementation, or maybe even an entire component, simply can not cope with a desired change. Patching at all cost component code to force the change is clearly inadequate compared to rewriting the component from scratch.

7. Avoid hidden assumptions and design decisions in the source code

Some assumptions or design decisions become hidden in the source code. When they change, it is easier to rewrite the entire component rather than to change the assumption in the existing code.

9.6 Product Line Architecture Techniques

9.6.1 Architecture Recovery

In software product lines, the products are derived from a common product family framework. Moreover, identifying the commonalities and variable parts of the products helps to define the product line architecture. This requires to compare the architectures of several products using architecture recovery.

Architecture recovery is a process that extracts architectural models from the source code, to analyze the dependencies among the components and redocument the architecture of an existing software.

Various methods have been proposed to assist architecture recovery such as the Software Bookshelf [FHK+97], the Dali workbench [KC97],...
9.6.2 Architectural Design Model

Bosch has proposed a product line architectural design [Bos00]. In his model, the design consists of the following activities:

- **"Business case analysis"** evaluates if the application of this model will be economically valuable.

- **"Scoping"** defines the boundaries (products and features) of the product line.

- **"Product and feature planning"** proposes pro-actively new possible products and features in order to refine the scope.

- **"Design of the product-line architecture"** is divided in three steps:
  1. "*Functionality-based architectural design*" identifies and defines the core abstractions of the product line, taking into account variabilities.
  2. "*Architectural assessment*" evaluates the product line architecture against its quality requirements.
  3. "*Architecture transformation*" manages the product line architecture evolution, supporting three transformation aspects: variants, optionality, and conflicts.

9.6.3 Product Line Architecture Description languages

Architecture description languages (ADL) allow formal description of architectures. These languages provide notations to represent architectural structures such as components, connectors, systems and properties. In the following, we present two architecture description languages especially adapted to describe software product line architectures.

- Koala ADL

Koala is a component model which has been developed by the Phillips research laboratories [vO02] based on an architectural description language used to build a large diversity of products from a repository of components. The Koala model was inspired by Darwin’s theory of evolution, this is the reason why Koala components can be combined into compound components, which can be combined into (more) compound components until, ultimately, a product is constructed.

Various concepts are used in the koala components models, like interfaces, connectors, subcomponents and compound components. Indeed a component has two different kinds of interfaces, the first one is called "provides interface" and the second "requires interface". **Provides interface** allows the environment of the component to use functionality implemented within the component. **Requires interface** allows the component to use functionality implemented in the environment. All "requires interfaces" of a component must be bound explicitly when instantiating a component. The components can be connected via several connectors called "direct connector", "switch" and "glue module". Furthermore, components may contain subcomponents, in this case we speak in terms of "compound components" as shown in figure 7.
The variation points in Koala are usually represented by a design pattern defined by a switch connector which can be managed thanks to an external configuration management system.

- **xADL 2.0**

xADL 2.0 is a software architecture description language (ADL) developed by the Institute for Software Research at the University of California, Irvine. It’s mainly based on xArch [xAr01], a core XML schema defined jointly by UCI and Carnegie Mellon University. Thus, as many usual ADL, the objective is to model the architecture of software systems, but xADL 2.0 is quite original in the sense that this ADL is defined as a set of XML schemas. It brings several advantages like the opportunity to easily extend the architecture or to use XML tools.

**Optional elements** (components, connectors, or links) in the architecture can be instantiated or not, depending on a guard condition’s satisfaction. **Variants** are represented as union types. A variant type may take on one of many actual types depending on the guard conditions’ satisfaction.

Following [DvdHT02], ”The key contributions of xADL 2.0 are its set of modular XML schemas that provides a basic framework for modeling product family architectures, its extensibility to allow future additions to (and modifications of) elements in the representation, and its associated tool support to automatically generate APIs for manipulating specific instances of product family architectures.”

### 9.7 Modeling variability in product line architectures

In this section, we insist on the variability aspect in the product line architectures. We focus on two interesting approaches modeling variability in architecture. The first
approach, proposed by Clauss in [Cla01], is based on UML stereotypes and Class Diagrams. The second, proposed by Keepence and Manion [KM99], is using patterns to model variability dealing mainly with class diagrams and the notion of discriminants.

9.7.1 Product Line Architecture Variability and UML

One purpose of the Product Line Architecture (PLA) is to identify variability and to provide built-in mechanisms to achieve it. One way to model product line variability in a architecture is to use UML class diagrams (see figure 8). Clauss proposes in [Cla01] to model variability using those concepts:

- Each variation point is described by a class with «variation point» stereotype. Additional tagged values are used like the specification of a binding time chosen between {development, installation, runtime} or multiplicity (i.e. how many variants can be bound at binding time).

- Each variant is described by a class with a «variant» stereotype and a condition tagged value. This condition specified when the variant has to be included or not.

- The relationship between a variation point and its variants is simply modelled by the classical inheritance relationship.

- Interactions between variants are modelled using dependencies “mutex” and “requires”. Furthermore, the stereotype «optional», tag values and the condition are used to describe optional elements.

![Figure 8: Order process system [Cla01]](image)
9.7 Modeling variability in product line architectures

9.7.2 Product Line Architecture Variability and Discriminants

To support commonality and variability in product line architectures, Keenence and Manion propose to combine patterns and discriminants [KM99]. A pattern is usually represented by class diagrams and object diagrams. A discriminant is any features (or requirements) that distinguish one system from another. There are three basic types of discriminants [KM99]:

- **Single discriminants** are a set of mutually exclusive features.
- **Multiple discriminants** are a set of optional features that are not mutually exclusive.
- **Option discriminants** are single optional features that might or might not be used.

The **single discriminant** can be represented as an inheritance hierarchy, in which commonalities are described in a base class and variabilities correspond to subclasses. Only one subclass can be instantiated in a system. The set of subclasses is called **realm**. New variabilities can be added extending the realm (see figure 9).

![Figure 9: Single discriminant [KM99]](image)

The **multiple discriminant** can also be represented as an inheritance hierarchy like the single discriminant. However, now, more than one subclass can be instantiated in a system. Each subclass has one single instance and the set of instances is held in a **collection** (see figure 10).

The **optional discriminant** is represented by two classes associated by a 0-1 relationship on at least one end (see figure 11).
The notion of component is defined 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" [Szy98].

Reusable components must be designed to be robust and general without loss of performance. In order to develop such components based on the product line architecture, three different opportunities are offered to companies. Firstly, components can be developed internally, from scratch, by the company’s developers. Secondly, components can be internally mined from already existing assets. Thirdly, companies can launch offers to bought COTS and integrate them in their existing software product line.
10.1 Mining existing assets

"Mining existing assets refers to resurrecting and rehabilitating a piece of an old system to serve in a new system for which it was not originally intended" [CN01].

The components rehabilitation consumes many resources and needs good documentation support. In the software product line context, the rehabilitated components must reach a high level of quality. For us, the mining existing assets solution must be avoided as much as possible in the software product line context. It is preferable to redevelop the component from scratch.

Some strong conditions are required to mine an existing asset for a software product line:

- The asset is clearly and completely documented.
- The asset is easily extensible.
- The asset needs little or no change.
- The asset can be easily related to the reference requirements.
- The asset can be easily integrated to the product line architecture.

A classical example is "wrapping an asset" which allows to add a new layer above the component, without modifying it, in order to change the interface which must comply to a new architecture.

In order to assist the mining of existing assets for software product line, one main method has been proposed by Bergey. Options Analysis for Reengineering (OAR) is "a method that evaluates the feasibility and economy of mining existing components for a product line" [BOS01].

10.2 Commercial Off-The-Shelf

A Commercial Off-The-Shelf product or COTS is defined in [CeB03] as a software product,

- developed by a third party (who controls its ongoing support and evolution),
- bought, licensed, or acquired for the purposes of integration into a larger system as an integral part, i.e. that will be delivered as part of the system to the customer of that system (i.e. not a tool),
- which might or might not allow modification at the source code level,
- but may include mechanisms for customization,
- and is bought and used by a significant number of system developers.

When you want to externalize the development of software product line components, you must focus on three main points:
You need to describe and define completely the requirements and the variability concerning the component. Based on these documents, you can invite tenders on the free market.

You need to evaluate the COTS, firstly, to compare the different solutions proposed by various companies and, secondly, to test the component when you receive it.

You need to integrate and maintain this new external component. Thus, two possibilities are offered to you:

- you have a complete access to the code and you maintain it by yourself.
- you do not have access to the code and you must have the guarantee that the vendor is stable and will maintain the component.

In order to assist the COTS selection and evaluation, a method has been proposed by the SEI. "The COTS-Based Systems (CBS) initiative is a set of practices to help with COTS product and technology evaluation, COTS acquisition and management, design and software engineering using COTS products, business-case analysis, and COTS policy and planning" [CN01].

11 Traceability

"Traceability is the ability to document and follow the life of a concept throughout system development. It is forward directed (post traceability: describing the deployment and use of a concept) as well as backward directed (pre traceability: describing the origin and evolution of a concept)" [ABFG00].

Traceability plays a key role in the development of software systems by confirming the compliance of a system with its requirements. Standards and guidelines for systems and software engineering identify requirements traceability as a major concern in achieving quality [GF94].

The central role played by traceability in software development does not need to be proved once again. Indeed, maintaining a link between requirements, their sources and the various artifacts developed during the system development life cycle is clearly useful to

1. Demonstrate how each requirement has been satisfied
2. Manage changes
3. Estimate costs for evolutions/changes
4. Facilitate communication among the various stakeholders

In the software product line context, adding or changing requirements can cause a degeneration of the product line architecture. To avoid this degeneration, information about dependencies and traceability have to be included into models. Software product lines traceability is mainly defined between the core assets (reference requirements \(\leftarrow\) reference architecture \(\leftarrow\) reusable components).
Moreover, traceability in software product lines is more complex and critical compared to classical software development traceability. Three main reasons explain this situation; first, changing common requirements generates automatically impacts on several products and core assets, secondly, the requirements traced do not necessarily exist in all products and, thirdly, a specific implementation may be linked with various products.

In [BW01], the authors propose one approach to introduce traceability in software product line context. First, they define the minimal traceability requirements they need:

1. It should be based on the semantics of models that are used in the product line infrastructure.
2. It should be customizable in terms of available trace types.
3. It should be capable of handling variability in the product line infrastructure.
4. The number of traces should be as small as possible.
5. It should be automatable.

Secondly, they describe two steps to add explicit traceability support to existing product line methods:

1. Defining potential links in a metamodel.
2. Creating models and establishing traceability links based on the metamodel.

Thirdly, they illustrate this approach with the PuLSE [BFK+99] method that we will present further.
11 TRACEABILITY
Part III

Software product lines researches and applications

12 Software product lines projects

The software product lines’ paradigm has been studied for several years now. Different projects and industries’ applications have been accomplished. In this section, we present different representative projects, describing their objectives and expected results.

12.1 ARES

12.1.1 Objectives

The main objective of ARES [Are99] (or “Architectural Reasoning for Embedded Software”) is to “enable software developers to explicitly describe, assess, and manage architectures of embedded software families”. To reach this goal they select, extend or develop a framework of methods, processes and prototype tools to incorporate architectural reasoning along the life cycle of software product lines. The research is conducted focusing on four aspects:

1. Specification of software architecture
2. Architecture recovery
3. Architectural assessment and analysis of software
4. Development and evolution of family architectures

12.1.2 Results

Results of this project will help to design reliable systems, which satisfy important quality requirements, which evolve gracefully and which may be built in-time and on-budget. The developed method will be incorporated in the software development life cycle at ABB, Nokia, and Phillips.

12.2 PRAISE

12.2.1 Objectives

The purpose of PRAISE [Pra00] (or “Product-line Realization and Assessment in Industrial Settings ESPRIT”) is to assist organizations in evolving their software development practices into a product line approach. The project wants to provide integrated and validated methodological support to the development of large software product families in industrial settings.
12.2.2 Results
The project will result in the definition of a software process, including domain and application engineering processes to develop software-intensive product lines. This process will include techniques for the development of domain-specific architectures, means to express variability and commonality between systems within a product-line, and traceability between architectural decisions and functional and non-functional requirements. This software process will be incorporated into two real-scale industrial experiments such as vehicle simulators (Thomson CSF) and car periphery supervision sensors (Bosch).

12.3 ESAPS

12.3.1 Objectives
"ESAPS (or Engineering Software Architectures, Processes and Platforms for System-Families) aimed to provide an enhanced system-family approach to enable a major paradigm shift in the existing processes, methods, platforms and tools" [Esa01].

12.3.2 Results
The ESAPS project develops concepts for product family engineering, based on its core process. The official results defined by the ESAPS project are:

1. Enhanced system family engineering processes.
2. Enhanced system family engineering methods and tools.
3. Enhanced component based domain specific platforms for system families.
4. Requirements for engineering tool suppliers to adapt their tools to comply with system-family engineering needs. Resulting from 1 and 2.
5. Requirements for generic and domain-specific middleware suppliers. Resulting from 3.

And practically, the partners will validate the viability of the enhanced approach in the various application domains such as healthcare, air supervision, communication, public utility management, automotive and office equipment support.

12.4 CAFÉ

12.4.1 Objectives
The CAFÉ [Caf03](From Concept to Application in System-Family Engineering) project continues the work where the ESAPS project has stopped. The purpose is to extend the product line concepts developed in the ESAPS project and transferring them into industrial practice. This project aims to considerably improve the European competitiveness in the area of industrial software development for product families.

12.4.2 Results
The CAFÉ project aims to gather knowledge about methods and procedures in order to practically apply the results defined in the ESAPS projects. A more long term perspective will be to develop tools based on these methods and procedures.
13 Software product lines development methods

Encouraged by the increasing success of the software product lines approach, several methods have been proposed, during the recent years, focussing on product line oriented software development and maintenance. In this section, we introduce well-known methodologies designed to develop and deploy product lines beyond existing domain engineering approaches. We do not present Aspect Oriented Programming methods for software product lines which constitute, for us, a different research axis. Nevertheless, details about the connection between aspect oriented technologies and software product lines technologies can be found in [BCS’01], [BLHM02], [KN00].

13.1 FAST

Family-oriented abstraction, specification and translation (FAST) [Wei95] is a product line engineering method proposed by Weiss from Lucent. This method focuses on providing a systematic approach to analyze potential families and to develop facilities for efficient production of family members. This method identifies the need for creating abstractions that remain stable during the evolution of the product line.

FAST is on of the first method which describes the classical systematic product family process with its two sub-processes (domain engineering and application engineering) and their feedback-loops [Wei95]. During domain engineering, the goal is to define the family and to develop an application engineering environment for producing family members. In this sub-process, requirements for the product family are defined using the "commonality analysis" [Wei97] and the reusable assets required for building the family members are developed. During application engineering, the goal is to use the reusable assets to produce family members.

Furthermore, the new idea [GJKW97] is the development of a domain specific language AML (or Application Modeling language), that implements the assumptions which were discovered during domain analysis. With AML, the requirements can be specified and can automatically be translated into individual products with an associated compiler.

Nevertheless, the requirements definition and specification are too complex for organizations as they need to be extremely precise for the code generation. Furthermore, problems with requirements management occurred rapidly, as the traceability aspect has been neglected.

13.2 FODA

FODA (or Feature-Oriented Domain Analysis) has been developed by Kang [KCH+90] and is composed of three main phases: domain analysis, feature analysis and feature modelling. Domain analysis focuses on scoping and identifying the products forming the product line. Feature analysis applies "commonality and variability analysis" to develop a list of common functions across the domain and a list of variabilities. This resulting lists are used to populate the feature model. Feature models consist of feature diagrams that represent features in a hierarchical structure. Mandatory features are those requirements of the system that have to be included in the product family. Alternative features are specializations of more general features which have multiple
specialization features, from which one has to be chosen. Optional features model requirements which can be included into one product, but may be left out of another. Variabilities between applications are captured in refinements.

The FODA method is described as a succession of steps:

1. **Context analysis** is relative to the domain’s identification and to the construction of a context model. A context model defines subsystems, which can be economically reusable, and gathers the product line requirements from various sources.

2. **Domain modeling** is relative to the selection and analysis of examples, needs and trends in order to identify essential commonality and variability. The product line requirements are modeled and analyzed using this identification, factoring and clustering of feature sets. A feature diagram is constructed describing the domain entities and the structural relationships among them by structuring and relating feature sets.

3. **Architecture modeling** derives, from these feature sets, an architecture model (also known as a design reference model) which can be instantiated to develop individual applications.

The FODA method can be useful for the requirements analysis, especially in capturing the feature models variability and commonality. Nevertheless, important guidance is needed and no solutions are proposed concerning requirements specification, management and verification.

### 13.3 FORM

FORM is a Feature-Oriented Reuse Method with domain-specific reference architectures developed by Kyo C. Kang and al. [KLD02]. FORM is an extension of FODA which aims to cover domain analysis and core assets development, and which aims to keep a business view on the software product line development. In FODA, the concept of using a feature model for requirements engineering was introduced. FORM extends FODA to the software design and implementation phases, and prescribes how the feature model is used to develop domain architectures and components for reuse.

FORM starts with an analysis of commonality among applications in a particular domain. The model constructed during the analysis is called a “feature model”, and it captures commonalities as an AND/OR graph, where AND nodes indicate mandatory features and OR nodes indicate alternative features selectable for various applications. Then, this model is used to define parameterized reference architectures and appropriate reusable components instantiated during application development.

### 13.4 RSEB

RSEB or Reuse-Driven Software Engineering Business [JGJ97] is a reuse process designed to accomplish key business goals and improve business performance [GFd98]. The idea is to center the approach on use cases, describing the product line requirements, elaborating the architecture and then bringing out object models directly linked with these initial use cases.
Explicit variation points and variants are defined in use cases and object models using extension of the Unified Modelling Language (UML). The reuse development process is defined following various steps:

1. **Requirements Engineering**: variability is captured by structuring use cases. High-level use case model describes the system context.

2. **Architecture Family Engineering**: A layered architecture is developed.

3. **Component System Engineering**: Reusable components are developed.

4. **Application System Engineering**: Applications are developed.

### 13.5 FeatuRSEB

FeatuRSEB [GFd98] or Featuring RSEB is a combination between the FODA method [KCH+90] and the Reuse-Driven Software Engineering Business (RSEB) method [JGJ97].

The idea is to add two more steps based on FODA to initiate the RSEB process: the domain engineering and feature modelling steps.

RSEB provides no explicit feature models, nevertheless, RSEB uses features informally as use cases or parts of use cases. Moreover, a new feature notation is proposed. The FODA feature diagram is changed in a tree or a network of features which are linked together by UML dependencies or refinements. Reusing the idea proposed in RSEB, the variation points are explicitly represented. This feature model is used as a "roadmap" for other RSEB models, guiding the product line engineers.

### 13.6 ConIPF

ConIPF or Configuration of Industrial Product Families is European FP6 project which wants to integrate both the product line approach and the structure-oriented configuration technologies [TKG02]. A methodology is developed top-down, starting from requirements defined through case studies of industrial partners and will be validated through concrete applications. The main idea is to apply and adapt configuration methodologies developed in artificial intelligence for the product lines context focusing on the development with reuse.
13.7 PuLSE

The PuLSE (Product Line Software Engineering) method [BFK+99] is designed to support customers in developing product lines and has been developed by the Fraunhofer Institute Experimental Software Engineering (IESE). This methodology claims that the development of a software product line must focus on the product itself and not on the organizational issues. The result is based on the IESE experience in new technology transfer for industry. This methodology seems interesting in our context as the authors argue in [KMSW00] that this method is not only relevant for big companies but also for small and medium companies. Thus, we want to describe more precisely this method which can offer a good base for the future methodology wanted in the PLENTY project.

The global philosophy of this method can be described as it follows:

- PuLSE provides a complete framework that covers the whole software product line development life cycle, including reuse infrastructure construction, usage, and evolution.

- PuLSE is modular and customizable: It consists of six technical components that can be selected and instantiated in order to satisfy the needs of specific companies.

- PuLSE can be introduced incrementally by augmenting existing software development processes and products with product line specific aspects step by step.

Basically, PuLSE is composed of three main elements; the deployment phases, the technical components and the support components as shown in figure 12. The main idea is that, during the deployment phases, the technical and support components are used and customized in order to elaborate a specific software product line.
13.7.1 Deployment Phases

The deployment phases describe the activities performed to set up and use the product line. They describe the four life cycle stages of a software product line infrastructure:

1. Initialization

Initialization implies studying the company and customizing PuLSE as a result. This phase is divided in three parts: baselining, evaluation and customization:

- Baselining: Regroup all the characteristics of a product line, called customization factors, that can have an impact on the development of the product line. First, relevant customization factors (for example type of applications or amount of resources needed) are selected using the PuLSE support components. Then, values for the selected factors are determined following strategies. The current profile records the values determined for the company.

- Evaluation: Analyze customization in order to take decisions for the customization. The factors, however, are not independent as some factors influence others. A decision tree of factors captures these dependencies and helps to determine the effects on the PuLSE components. The result of
this step is used by the component PuLSE-CDA. When existing baselining profiles match the current profile of the company, information from the previous experience can be reused for evaluating and customizing.

- Customization: apply the decisions taken in the evaluation to determine a complete process including the definition of work products, their relations and their representations used and based on the decisions made during evaluation.

2. Infrastructure Construction

Infrastructure construction implies a scope model and the definition of the product line architecture. This complex task is assisted by three technical components: PuLSE-Eco which helps to determine an economic viable scope for the product line, PuLSE-CDA which is used to elicit and articulate product line concepts and their interrelationships, and PuLSE-DSSA which is applied to define a software reference architecture for the product line.

3. Infrastructure Usage

Infrastructure usage implies the use of the infrastructure to create product line members. The usage phase of PuLSE aims to specify, instantiate, and validate one member of the product line. If changes or extensions to the product line are necessary to build an infrastructure instance, they are passed as change requests to PuLSE-EM. The customer requirements are used to plan the development process for the instance. Then the product is specified and validated against the customer requirements.

4. Evolution and Management

Evolution and Management imply the evolution of the infrastructure and the way to manage and coordinate it through the three previous phases.

13.7.2 Technical Components

PuLSE consists of six technical components that provide the technical know-how needed to operationalize product line development. In particular, the technical components relevant to the core technical activities (PuLSE-Eco, PuLSE-CDA, PuLSE-DSSA, PuLSE-I) are adaptable to the various situations and deployment phases. However, in full product line applications they are typically applied in a specific order.

1. PuLSE-BC

PuLSE-BC (Baselining and Customization) is used to create an instance of the PuLSE method that is tailored to a specific company context. Here is the detailed process in figure 13.

2. PuLSE-Eco

PuLSE-Eco (Economic Scoping) is concerned with an economic analysis of the product line. The goal is to identify candidate reusable assets that provide a high
economic benefit to the organization at low risk. It defines the economic scope of a product line that draws upon existing work. PuLSE-Eco is divided in four phases:

- Map out product candidates which determine anticipated product line members and map out their characteristics paying attention to the business objectives and to evaluation functions.
- Characterize products, apply the characterization functions on the product characteristics.
- Benefit analysis
- Develop the product line plan

The detailed process is illustrated in figure 14:
The **scoping purpose** is to find the feature set of products that should be included in the product line to maximize the benefits for the product line producer. The input for the method is provided by the stakeholders and by the marketing which provides information about the market and its customers. The scope definition identifies the variabilities and commonalities of product line members and also assigns priorities to each features. This scope identifies the existing, future and potential products and their characteristics within the product line. Two types of evaluation functions are used to define the scope, first the *characterization functions* and then the *benefits functions*.

The scoping method is influenced by the producer benefits, and leaves to the method’s user the responsibility to include the correct stakeholders to address the customer perspectives. Moreover, the method focusses more on products then on the customers.
3. PuLSE-CDA

PuLSE-CDA (Customizable Domain Analysis) is a domain analysis approach that can be modified following the various characteristics of an organization. This technical component is essentially composed of requirements and knowledge engineering techniques.

The initial domain descriptions (in terms of features) serve as a starting point. Domain analysis in PuLSE is customizable, in the sense that the specific work products which are produced by domain modeling may vary, depending on the domain (i.e., the specific views on the domain that are produced and the specific notations used to represent them are variable).

Given the feature descriptions produced during scoping as a main input, domain information elicitation is started to produce a domain model. Information can be extracted from textbooks, experts, pre-existing documents, etc...

In addition to the work products, a domain model also contains a decision model that captures the knowledge needed to instantiate the domain model. A decision model is built on top of the variation points in the work products. It consists of a decision hierarchy, which is grounded on simple decisions that capture the rationale and the possible choices for a single point of variation. Dependencies among decisions are explicitly captured by constraints. Decisions, are described through a description of the decision, a description of decision types (optionally, alternative, group), and constraints linking this decision to other decisions in the decision model.

The detailed process is illustrated in figure 15.
Figure 15: PuLSE-CDA process [BFK+99]

4. PuLSE-DSSA

"PuLSE-DSSA (Domain Specific Software Architecture) supports the definition of a domain-specific software architecture, which covers current and future applications of the product line as described by a product line model" [BFK+99]. PuLSE-DSSA is used to develop a **reference software architecture** for all products in the product line. The focus is on architectures that are implementable in terms of object-oriented frameworks, and on methods for analyzing the correctness, quality, traceability and evolution of architectures. The architecture is thus developed incrementally guided by the **generic scenarios** following the process shown in figure 16.

The inputs of this process are the scope definition and the domain model, including a decision model that describes the dependencies and inter-relationships among variabilities in the domain model.

The scope definition is provided by PuLSE-Eco and is used during the step "Select scenarios and plan next iteration" to determine the domain importance of specific scenarios. The set of generic scenarios, that must be supported by the product line architecture, is derived from the generic storyboards and the domain model defined in PuLSE-CDA.
The detailed process is illustrated in figure 16.

Figure 16: PuLSE-DSSA process [BFK+99]

5. PuLSE-I

PuLSE-I (Instantiation) is concerned with product instance development, i.e., deriving a concrete product from the product line infrastructure. The key elements include instance planning, instantiation of domain model and reference architecture, as well as product construction, which is mainly based on the usage, adaptation, and extension of existing reusable components. Here is the detailed process in figure 17.
6. PuLSE-EM

PuLSE-EM (Evolution and Management) deals with product line evolution and management. This activity focuses on three main goals:

- Managing the instantiation of the PuLSE process,
- Defining feedback processes that allow continuous optimization of both the processes and the product line artifacts,
- Defining and realizing an appropriate configuration management strategy.

It describes how to integrate new products in the product line, and deals with configuration management issues as products accrue over time. Here is the detailed process in figure 18.
Figure 18: PuLSE-EM process [BFK+99]
13.7.3 Support Components

The Support Components regroup the entire documentations useful for a better adaptation, experience capitalization, evolution and installation of the product line.

1. Project Entry Points

Project entry points customize PuLSE to predefined project types. “The PuLSE project entry points describe standard situations where PuLSE can be applied”[BFK+99]. Thus, it helps to integrate PuLSE into a specific company context. Three entry points already used are [BFK+99]:

- **Pure PuLSE**: It characterizes the situation where a new product line is set up in a company. In that context, all PuLSE components can be established with full traceability.
- **Evolutionary PuLSE**: The components of PuLSE can be integrated one by one in a currently running development process.
- **Reengineering-driven PuLSE**: Existing assets are reused to enrich the PuLSE component deliverables.

2. Maturity Scale

Maturity scale provides an integration and evolution path for product line adoption to companies using PuLSE. “The PuLSE maturity scale is designed to help a company to adopt the methodology step by step”[BFK+99]. It drives towards the usage and integration of the PuLSE phases and components. It measures the company’s product line level. The various possible levels are described in [BFK+99] as:

- **Initial**: Single PuLSE technology components can be applied independently of one another – mainly Eco, CDA, DSSA after the necessary customization.
- **Full**: All components participating to the Construction Phase are used, yet their degree of integration can vary substantially.
- **Controlled**: PuLSE is applied as a complete development life-cycle. Traceability (integration) among phases is established and maintained.
- **Optimizing**: The PuLSE development life-cycle is refined over a number of instances using optimization techniques.

3. Organization Issues

Organization issues provide guidelines to set up and maintain the right organization structure for developing and managing product lines. Several guidelines have been proposed in [BFK+99]:

- **Developing guidelines**
  - Subdivision of the application domain into areas of specialization according to separation of concerns.
Vertical layering decomposition of the development into the application area analysis, architecture, implementation and deployment.

Assignment of permanent personnel to these areas of specialization within the layered decomposition whenever possible.

Supervision and enforcement of the process rules, architecture soundness and evolution.

- Managing guidelines
  - Dynamic assignment of personnel to projects according to their responsibility area.
  - Very small project leadership. Project leaders coordinate the project development with the product line.
  - A developer assignment ratio of 70% to current project development and 30% on the evolution of the reusable infrastructure seems to be appropriate.

### 13.8 KobrA

The KobrA [ABB⁺01] or “Komponentenbasierte Anwendungsentwicklung” or “component-based application development” method has been developed at the Fraunhofer Institute for Experimental Software Engineering (IESE). From the practical point of view, KobrA can be viewed as a “ready to use” customization of PuLSE. This systematic method is designed for component-based product line engineering. The key innovation is the integration of the product line and component-based approaches. The idea is to unify in this method the product line paradigm supporting “reuse in large” and the component based paradigm supporting “reuse in small”. Indeed, Colin Atkinson and al. claim that “the product-line and component-based approaches to software development seem to have complementary strengths. They both represent powerful techniques to support reuse, but essentially at the opposite ends of the granularity spectrum” [ABM00].

Thus, the KobrA method is principally divided in “framework engineering” and “application engineering” activities. The framework engineering activity is based on the product line approach in order to design and maintain a generic framework describing variabilities and commonalities. A framework is defined as the static representation of a set of KobrA components (Komponents) organized in the form of a tree. The results generated by this activity are framework models defined in terms of a mixture of textual and UML-based models.

The application engineering activity is based on the component based approach in order to instantiate the generic framework. Each Komponent is described at two levels:

1. **Specification level**, which defines the Komponent’s externally visible properties and behaviors,
2. **Realization level**, which describes how the Komponent is decomposed in lower level Komponents.

The final goal is to develop applications containing specific variants corresponding to particular customers’ requirements. The results generated by this activity are application models defined in terms of a mixture of textual and UML-based models.
13.8.1 Framework engineering

Framework engineering is divided into four activities:

- Context Realization
  The purpose of this process is to elicit the environment properties and to define the framework scope. This phase takes also into charge the variabilities and commonalities analysis. The *business process model* and the *decision models* are produced.

- Komponent Specification
  Using the business model and the decision models, the purpose of this process is to describe the externally visible properties of a Komponent. To fulfill this goal, the *structural model* (UML Class/Object diagrams), the *behavioral model* (UML Statecharts diagrams), the *functional model* (operation schemata), and the *decision model* (textual) are produced for each Komponent. "An operation schema defines operation effects in terms of input parameters, changed variables, output values as well as pre- and post-conditions" [Ca94].

- Komponent Realization
  The purpose of this process is to describe the private design of a Komponent realizing specification. To fulfill this goal, the *interaction model* (UML collaboration diagrams), the *structural model* (UML Class/Object diagrams), the *activity model* (UML activity diagrams), and the *decision model* (textual) are produced for each Komponent.

- Component Reuse
  The purpose of this process is to reuse pre-existing components or to integrate directly executable COTS components. The main task is to manage a process of negotiation between the reusing Komponent (desired specification) and the reused component (offered specification) in order to define a mutually acceptable specification.

13.8.2 Application engineering

The application engineering process uses the different models defined in the framework engineering in order to recursively transform all the generic framework models for the particular application. During this phase, the framework built during framework engineering is used to construct specific applications.
The application engineering is divided into two activities:

- Context Realization instantiation

  The purpose of this process is to instantiate the framework’s context realization. The potential users’ problems are analyzed and compared to the features supported by the framework. To fulfill this goal, context decisions and a concrete realization of the application’s context are produced. To finish this process, customer-specific requirements can be added and the realization of the application context must be evaluated concerning its completeness and its correctness.

- Framework Instantiation

  The purpose of this process is to instantiate recursively the generic Komponent hierarchy of the framework using context decisions and removing unwanted features.

13.9 Carnegie Mellon University Framework

In this section, we introduce the Framework for Software Product Line Practice Version 4.1 [CN03] from the Carnegie Mellon University. This framework is developed for the software product line context, giving many indications concerning the essentials of software product lines. This framework is based on Clements Book [CN01] and intends to evolve over time, gathering information relative to the lessons learned by companies using the product line approach.

The official goals [CN03] of this framework are:

1. to identify the foundational concepts underlying software product lines and the essential activities to consider before developing a product line.
2. to identify practice areas that an organization developing software product lines must master.
3. to define practices in each practice area, where current knowledge is sufficient to do so.
4. to provide guidance to an organization about how to migrate to a product line approach for software.

Such a framework raises the level of abstraction and is compatible with a large number of specific development strategies but is not ready to use “out of the box”. This framework must be tailored to the needs of specific projects.

The first part of this framework introduces the notion of software product lines and presents the various costs and benefits relative to it. The second part defines the three essential processes implicated in a product line approach: Core assets development, Products development and Management. In the third part, the various practice areas are described differentiating the software engineering practice areas, the technical management practice areas and the organizational management practice areas.
This framework is an essential reading, in the sense that it proposes a large and detailed overview on most product line aspects. Each topic is presented taking into account the traditional approach and insisting on the aspects peculiar to product lines. Those topics are also described with the core assets development perspective and with the products development perspective. Furthermore, for each topic, specific practices are proposed and practice risks are considered.

Many sections included in this framework are important and vital to elaborate an efficient software product line. Nevertheless, as we focus on software development methodologies, we are more interested in the section dealing with software engineering areas, where the classical software development steps (domain understanding, requirements engineering, architecture definition, architecture evaluation, component development, cots utilization, mining existing assets, testing) are introduced and studied considering the product line context.

14 Software product lines applications

Software product lines concepts are applied to many different application domains where software is needed: telecommunications, home appliances, car industry, etc...

In the following, we shortly describe four representative applications in various domains. We want to underline here that a constructive reuse can be applied by many different organizations considering their own specific context.

14.1 Car Periphery Supervision (CPS)

Car periphery supervision systems [TFF+01] can be described as a family of automotive products designed to guarantee more safety and comfort for car drivers. These systems are based on sensors installed around the vehicle to monitor its local environment.

Figure 19: Car Periphery Supervision systems [TFF+01]
The functionalities provided by these systems are parking assistance, automatic detection of objects in vehicles blind spots and the adaptive control of airbags and seatbelt tensioners.

**Commonalities** mainly concern data analysis. Indeed, each car periphery supervision system typically requests, filters, and evaluates data from sensors while at the same time performing diagnostics, availability tests, and consistency checks.

**Variabilities** mainly appear with sensor measurement methods and evaluation mechanisms.

### 14.2 Celsius Tech: ship system 2000

Ship system 2000 [BC96] can be described as a family of naval defense systems unifying all weapons systems, command-and-control, and communication functions on a warship.

**Commonalities** mainly concern communication, fire-control, electronic warfare,...

**Variabilities** mainly concern ships types (surface or submarine) and classes, weapon systems, languages, platforms, ...

### 14.3 Nokia: Mobile phones

Mobile phone systems offered by Nokia can be described as a family of systems offering telecommunication services.

The functionalities provide by those systems are, for example, messages, clock, alarm clock, countdown timer, calculator, currency converter, reminders, games ...

**Commonalities** mainly concern communication functionalities, usual accessory features, ...

**Variabilities** mainly concern number of keys, display sizes, languages, communication protocols, games, ...

### 14.4 Weather Stations

Weather stations systems [KS00] can be described as a family of systems measuring weather-related phenomena in the environment and transmitting this information.

**Commonalities** mainly concern sensor systems and algorithms to convert this sensor data into meaningful information, such as wind speed and direction. Moreover, weather station rules and constraints are highly standardized.

**Variabilities** mainly concern ways of communication and information formats (digital or analog). Furthermore, the environmental conditions (polar regions, equator regions, dry regions) cause strict and specific requirements on the system design. Indeed, many different kinds of weather stations exist such as ice weather stations, agricultural weather stations, high tower/NRC weather stations, water resource weather stations, shipboard weather stations, ...
14  SOFTWARE PRODUCT LINES APPLICATIONS
References


REFERENCES


