2.4. Software Product Line Engineering (SPLE)

Software Product Line Engineering (SPLE) is a paradigm that intends to develop software-intensive systems using platforms and *mass customization*[4] [PBvdL05]. The mechanism to achieve this is by identifying commonalities and variabilities in a set of products. Some examples of such artifacts are system requirements, architecture, components and tests.

Variability refers to the capability of an artifact to be configured, adapted, extended or modified for use in a specific context. The higher the variability of an artifact, the broader range of contexts this artifact can be used. Thus, the software product is more reusable. Management of variability encompasses the activities of representing variability in software artifacts throughout the lifecycle dealing with dependencies among different variabilities and supporting the instantiations of those variabilities.

SPLE paradigm separates two processes:

Domain Engineering. This process produces the platform including the commonalities of the products and the variability to support precisely mass customization.

Application Engineering. The process is responsible for deriving product line applications from the platform established in domain engineering.

2.4.1. Software Product Lines (SPL)

The goal of Software Product Lines (SPL) is to minimize the cost of developing and evolving software products that are part of a product family. Figure 2.10 illustrates this with an example of an organization that develops four products [SD06]. These products have common and variable features, marked with 60% and 40% respectively. If these products would be developed by separated, 60% of the cost for each product is spent on the same features, while the rest is spent on their differences. If the organization could develop some of the common features only once, the cost for developing the four products could be potentially reduced.

In SPL, variability can be understood by answering to three questions:

What does vary? -The software product that belongs to a product family.

Why does it vary? -There are many reasons why the software product is varying, like different information needs, technical reasons, etc.

How does it vary? -The software product is differentiated from the others based on the features that constitute them.

[4] Mass Customization is the large-scale production of goods tailored to individual customers' needs. [Dav87]

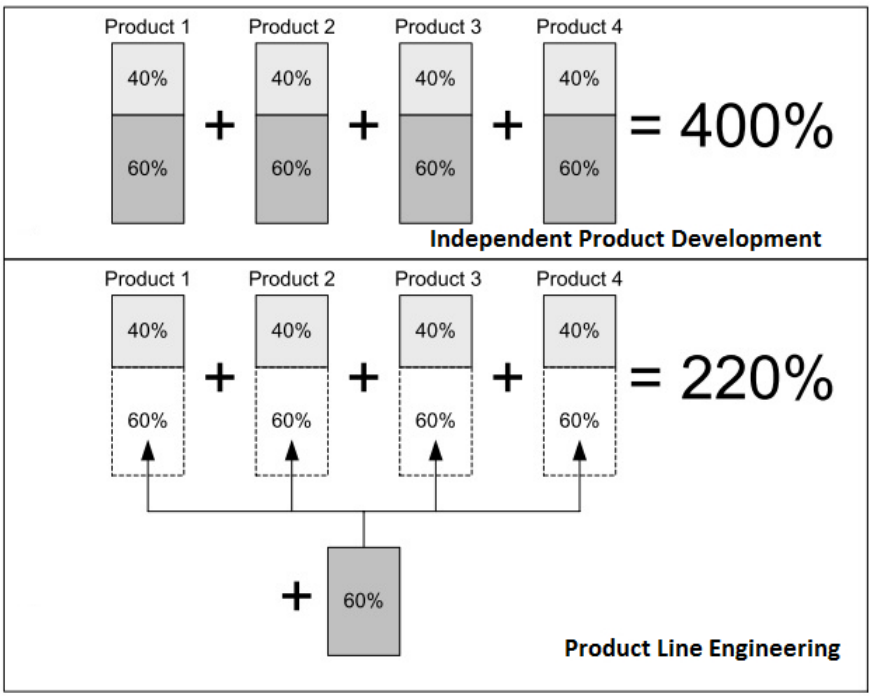

Figure 2.10.: Comparison of software product development

Taking as reference the previous example, depicted in Figure 2.10, regarding the four products and only for illustrating purpose, it is assumed that the product is a website that has many features. Among these many features, there is one feature called *payment method*. The website product can support in its feature of payment method a *payment by credit card*, *payment by cash* or both of them.

Since the differences between the products are described in terms of features, the concept of feature is introduced. Feature is a logical unit of behavior that is specified by a set of functional and quality requirements [Bos00]. A feature is a way to abstract from requirements. The features can be categorized according to their characteristics into three groups [GFd98].

Mandatory Features. These are the features that identify a product. That is, the commonalities among products.

Optional Features. These are features that can be part of a product and when enabled, add some value to the core features of a product.

Variant Features. A variant feature is an abstraction for a set of related features (optional or mandatory).

It is important to realize features and requirements are bound through an association. An association is an $n : m$ relation. This means that a specific requirement can apply to several features in the feature set and a specific feature can meet more than one

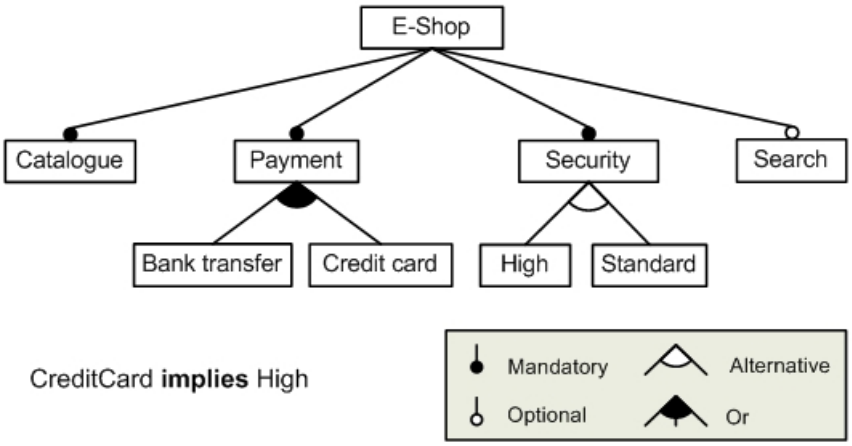

Figure 2.11.: A feature diagram representing a configurable system

requirement.

2.4.2. Feature Modeling

There exist methodical approaches such as Feature-Oriented Domain Analysis (FODA) [KCH+90] and Feature-Oriented Reuse Method (FORM) [KKL+98] that are used to model and represent the products of a SPL in terms of features. This representation is called feature model and it makes use of a conventional graphical notation.

A *feature diagram* is a visual notation that hierarchically structures the set of features from the model, which is basically a tree. The notation from the diagram can be matched with the feature categorization as follows:

Mandatory - a child feature is required.

Optional - a child feature is optional.

For the variant features:

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

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

In addition to the relationships between features, constraint dependencies are allowed. These constraints are *requires* and *excludes*. In the first case, and as an example *feature A requires feature B*, the selection of a specific feature in a product implies the selection of another feature that is bound by this constraint dependency. The exclusion, *feature A excludes feature B*, specifies that both features cannot be part of the same product.

As an example, Figure 2.11 illustrates how feature models can be used to specify and

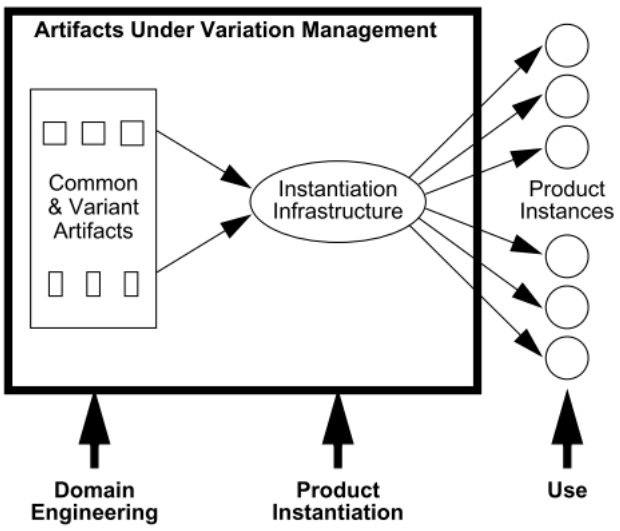

Figure 2.12.: Variation management of production line

build configurable website systems. The software of each application is determined by the features that it provides. The root feature (e.g. E-Shop) identifies the SPL. Every website implements a catalogue, a payment module, a security policy and optionally a search tool. Website must implement a high or standard security policy (choose one) and can provide different payment modules: bank transfer, credit card or both of them. Additionally, a constraint dependency forces websites including the credit card payment module to implement a high security policy.

One important aspect to notice from the feature diagrams is the locations at which the variation will occur. The locations are defined by Jacobson et al. as **Variation Point**. Any sub-feature that belongs to a variation point is known as a **variant** [JGJ97]. The main purpose of introducing a variation point is to delay a decision, but at some time there must be a choice between the variants and a single variation will be selected.

2.4.3. Variation Management

Software product line engineering and conventional software engineering have one big difference in between: variation management. Variation management in conventional software engineering deals with changes in the software over time and is commonly known as configuration management. Configuration management keeps track and controls this variation. Variation management in SPLE deals with variation not only in time but also in space and for that reason is commonly referred as multi-dimensional. In this context, managing variation in time refers to configuration management of the software product line as it varies in time, while managing variation in space refers to manage differences among the individual products in the domain space of a product

line at any fixed point in time [Kru01].

Variation management encompasses the activities of explicitly representing variability in software artifacts throughout the lifecycle, manages dependences among different variabilities and supports the instantiation of variants. Figure 2.12 shows variation management structured around a product line. In this case, only the common and variant artifacts and the instantiation infrastructure belong to variation management. The product instances are out of the scope from the variation management and are used directly without any further modification via application engineering. Domain engineering deals with the establishment of the platform and product instantiation refers to deriving the products. Product instantiation refers to application engineering.

At the beginning of the present section was mentioned that SPL deals with management over time and space. SPL uses different mechanisms for this kind of variation management. Versioning and branching mechanisms are used for time-based variation and variation point management deals with domain space. Following, a brief overview on these mechanisms is provided:

Version management. Sequential versions of files must be managed over time by operations like Check-in and Check-out.

Branch management. Independent branches of file evolution must be supported by creating parallel file branches.

Variation point management. Points of variation in the software artifacts of the production line must be managed by implementing the variation points in such way, that these include a collection of variants for selecting in the domain space.

2.5. Paper Prototyping

Paper prototyping is a group creativity technique developed to generate different ideas when designing, creating, testing and communicating any type of user interfaces (UIs). The resulting UIs are, later on, used for the selection of an appropriate solution. Examples of such UIs are software programs, applications from mobile devices and websites. Here is a formal definition of paper prototyping:

> *"Paper prototyping is a variation of usability testing where representative users perform realistic tasks by interacting with a paper version of the interface that is manipulated by a person 'playing computer', who does not explain how the interface is intended to work."* [Sny03]

The mechanism of paper prototyping starts with a meeting from all the members of the product team to choose the type of user who represents the target audience for the UI. Afterwards, a *usage profile* is created. Usage profile is a set of typical tasks describing uses for the system. The next step is to make screen shots and/or hand-sketched versions of all the components from the UI required for the usage profile.