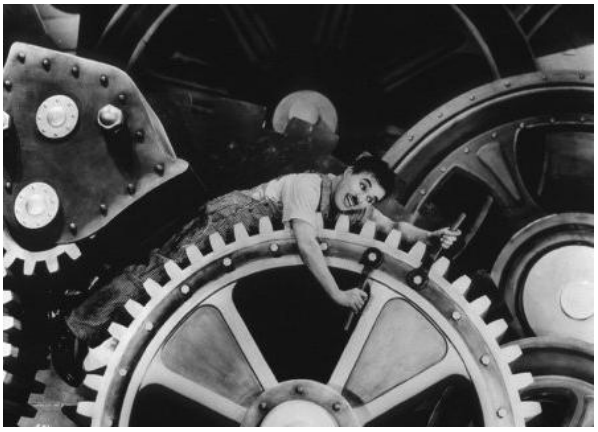
Blekinge Tekniska Högskola - A.Y. 2011/2012

Software Product Line in Practice



Case study : a product-line for static websites

Receiving teacher : Prof. *Ludwig Kuzniarz*

Group Members:
Farnaz Fotrousi [7712165062]
Carmine Giardino [870910T573]
Nicolò Paternoster [861218T753]

Group name : *Web-line*

Table Of Content

1. Introduction

//What is this?

Software Product-line in Practice is a white paper that provides a practical and pragmatic approach to product-line development, written by students, for students. There are many existing books that discuss product lines from different perspectives and with different approaches : in this paper we try to introduce the reader to product-line engineering providing as many practical example as possible. All the examples are taken from a project we worked for the course of *Product-line Architecture* [17], during the last study period.

//Web-line : case study

Web-line is the name we gave to our project, and it is basically a product-line designed to create two families of static web sites, letting the customer select among different options to customize the final product. The variety of different kinds of web-platform is huge: web-portals, news, personal sites, company sites, social networks and web-application are just a few of them. We narrowed our focus on static websites built in the domain of job seeking and offering. The common characteristics of suitable websites led us to build a product line which can generate two families of products : websites for companies and personal websites.

We use the data and the examples we collected during the construction of this project in each and every chapter of this paper, in order to give to the reader a *first-person* understanding of what the creation process of a product-line is like. We released our source-code under a GPL license. The project files are delivered with the present *pdf*, or available online at [15], while a deployed version of our *Web-line* can be tested at [16]. During the learning process the reader is free to take a look at the code and the product-line, considering that the artifact is still an *alpha* version.

//Document Structure

As you can see from the *Table of Content,* this document follows a top-down approach : we start by introducing the discipline of Software Product-line Engineering and we end up discussing all detailed processes and sub-processes [3] that are involved into it.

After introducing the *role of variability* in Software Product-lines, in chapter four we go through all the details of *Domain Engineering* : *Product Management*, Domain *Requirement Engineering* , *Domain Design* and *Domain Realization*. In chapter five we cover the issues related with *Application Engineering i.e. : Application Requirement Engineering, Application Design*, *Application Realization* and *Domain Testing.* In all these sections we enrich the discussion by providing the case-study of our *Web-Line* project. In chapter six (*Product-line Project Management*) we address managerial aspects and practices to build and maintain a good platform. In chapter seven we report and analyze some useful metrics related to our project. Finally, we summarize our experience in chapter eight (*Experience and Future Research*) and nine (*Conclusions*).

2. Software Product-line Engineering

From software productivity analysis , organizations can improve development efficiency reusing software artifacts instead of re-inventing the wheel for each product [13]. Despite it might look like a “silver bullet” [1], across dozen of studies, many researchers show that reusable artifacts cost about 50 percent more than one-off artifacts [2]. This statement might discourage undertaking this strategy in many organizations since their software projects are budgeted just to get their own job done and not for making it reusable. Nevertheless, introducing software product-line engineering might represent a good practice to overcome these problems using specific strategies described in this state of the practice.

2.1 Overview

Software product-line engineering (SPLE) is a software development paradigm that combines two main techniques: mass customization and common platform development. The former represents a large-scale production of goods tailored to individual customers’ wishes, whereas the latter is any base of technologies on which other technologies or processes are built [3]. By combining them, SPLE realizes the customers’ needs producing a common platform for the development of software products. The platform includes all common parts that suit all the products that are going to be developed and foster to determine the customers’ wishes for more customized products.

These two sentences respectively introduce two main concepts: commonalities and variability. Commonalities describe and identify the artifacts shared among the different products defined in the product-line platform. Variability describes and identifies where the *products* of the *product-line* may differ in terms of the features they provide. In the study of this project we will present at first how to find commonalities (common artifacts) and then we will define for each product (or family of products) customizable features.

To enhance the effectiveness of the commonalities, artifacts need to be produced in such a way that they can be reused in a future for developing new platforms. To ensure effectively this potentiality, three quality attributes need to be considered: maintainability, flexibility and evolvability. These three key concepts will be explained in detail in the next section related to the variability notion. This section focuses on the advantages and limitations of sharing a platform among several products. The main advantage achieved in making use of a product-line is the reduction of effort spent for creating and deploying a new product. It can be explained by the representation of the following figure.

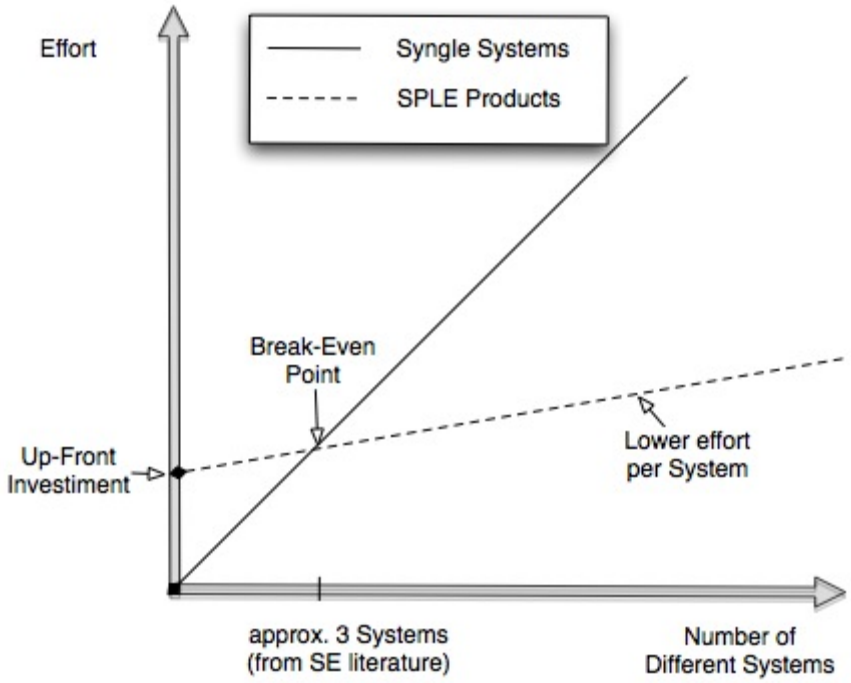


Figure 2.1 : Single systems vs SPLE

The figure compares "single systems" and SPLE products in terms of effort spent by implementing increasing number of different systems. What can be achieved by a proper implementation of SPLE is that by increasing the number of implemented systems, the effort is proportionally lower increasing in comparison to traditional single systems implementations. Indeed, the first investment of *up-front* effort, that is the activities for constructing the common platform, pays off when the number of systems are, statistically speaking, approximately more than three (in correspondence of the Break-Even point).

Other advantages can be summarized as follows:

- Reduction in number of defects per product for the reason that the artifacts derived from the platform development are reviewed and tested in more than one kind of product.
- Reduction of maintenance effort, and therefore reduction in the average engineering cost per product: the changes conducted in the platform development are propagated to all the products as well.
- Coping with evolution and complexity, and therefore growing in the total number of products that can be effectively deployed and managed. When needed, artifacts can be added to specific products to increase flexibility in the customization process.

Even though SPLE might be appealing in software development, many limitations need to be considered:

- Adequate technology for applying the principles of SPLE (i.e. object-oriented programming, component technology and binding techniques).
- Configuration management for complex systems, where to succeed in SPLE, many parts in different versions must be managed.
- Knowledge in the domain, that is important for being aware of identifying adequate commonalities and variability for developing platforms and products. At this regard, wrong choices of implementation are costly and might cause to experience much later pays off of the up-front investment (see *Fig.* 2.1).
- Domain stability, as important factor to evaluate the introduction or not of SPLE techniques: if software products changes in an unpredictable way, the commonality analysis is too costly

to be applied.

Explained the basic principles of SPLE, the framework of required processes are introduced.

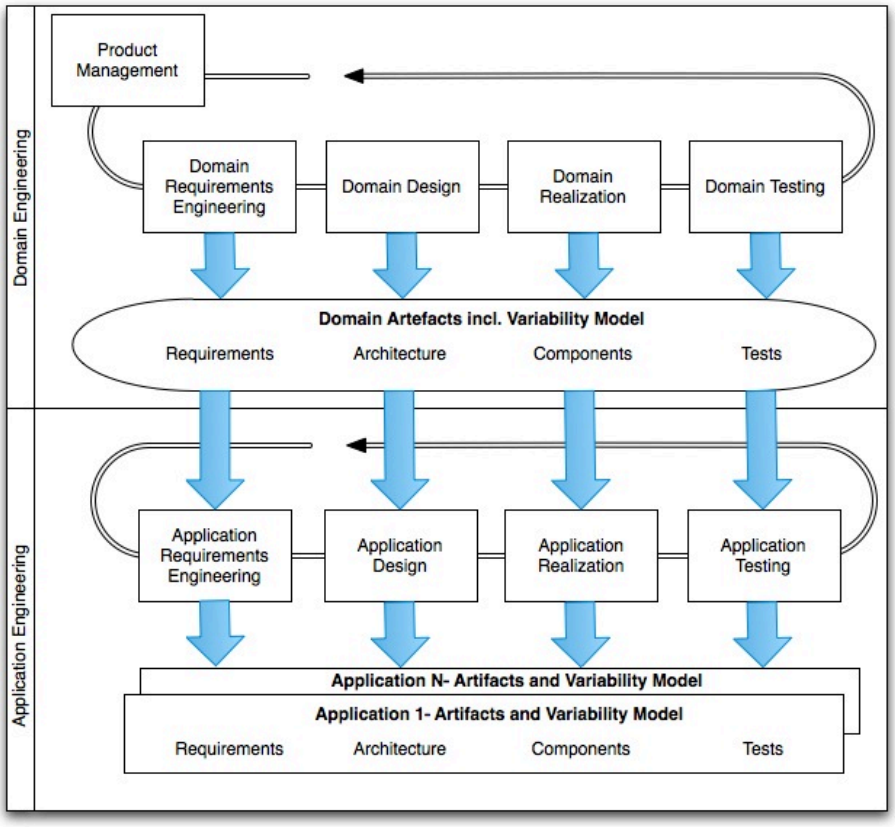


Figure 2.2: SPLE framework

The framework is divided in two main parts: domain engineering (also called core asset development) and application engineering (also called product development).
Both of them involves different sub-processes that are going to be explored in details in the following sections. In this overview is important to notice how the different sub-processes interact with each other and evolve during their development. From product management to domain testing the artifacts are produced in an iterative fashion, where each artifact represents an input to the sub-processes of application engineering. On the other hand, application engineering iteratively perform other sub-processes, which produce the final products/applications.

Product-lines have two different kinds of major stakeholders : on one hand there are the people involved in the construction of the SPLE and on the other the people that will finally make use of it. In the second category of stakeholders, we want to make a clear distinction between *customers* and *user*. With the term *customers* we refer to the stakeholders that directly interact with the *product-line*, whilst with the term *users* we refer to those who interacts with the final *products*. In the figure below we show a concrete example of such difference in our specific case .

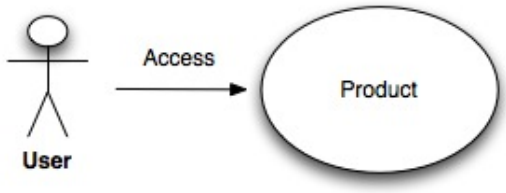


Figure 2.2a: User interacts with product

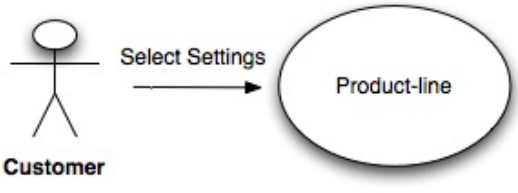


Figure 2.2b: Customer interacts with product line

The *customer* is the one who access to the product-line (*Fig. 2.2b*), selecting different options and configuring the product according to user preferences. The product is then accessed and utilized by the end *user* (*Fig. 2.2a*). We want to stress out this difference as much as possible, because we've found a general confusion and ambiguity in the use of these two terms, that have two actual different meanings. In the following we can remark that *products* are the output the product-line.

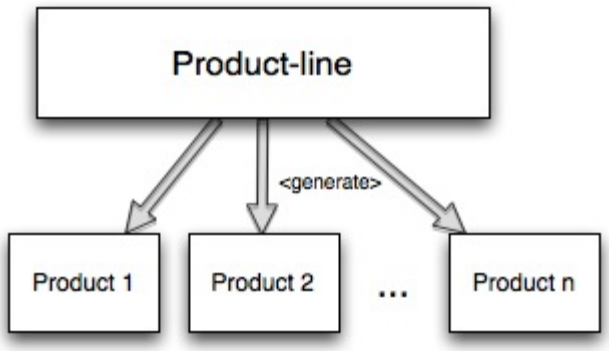


Figure 2.2c: Product-line and Products

As you can see from the *Figure 2.2c*, there is another important difference that we want to highlight for the sake of clarity, and is the difference between the *product-line* and its outputs, the *products*. One product-line can generate more than one product, or family of products. In this document we use the term *platform* for product-line and *application* for the product.

3. The role of variability in Software Product-line

In the common language the term *variability* refers to the capability or tendency to change. Through the analysis of its characteristics, variability can be synthesized in two elements: the *variability subject* and the *variability objects*. The subject answers the question *who* varies, whilst the objects answer the question *how* the subject might change.

3.1 Introduction

An real world example might be an attribute of an entity like the color of a shirt. The color represents

the *subject*, which might vary among several *objects* like: “blue”, “green”, “yellow” and so forth. To achieve a better knowledge of the variability phenomenon, it is important to stress also *why* a certain subject varies. It is relevant for understanding the reasons that can be related to stakeholders’ needs, country laws, technical factors etc. Define the rational behind the variability choices is even more significant in software environment where non-physical properties of the product increase the complexity of such decisions.

Moving forward into SPLE, the concept of variability acquires a central role for sharing a common and variable set of features. In this context, the variability subject and object terminologies are projected respectively into *variation point* and *variant expressions*. Then, the two expressions are going to be detailed with the definition and examples. *“A variation point is a variability subject within domain artifacts enriched by contextual information”* [3]

Each variation point needs to be related to one or more domain artifacts, *i.e.* the outputs of SPLE sub-processes defined in the previous section (*Fig. 2.1*), enriched by an explanation of reasons behind such variability.Within the defined domain artifacts, variants are shown as representations of one or more variation points. An example might be the operating system, that is a variation point, which variants could be a release for *Linux* or *MacOsX*.

3.2 Internal vs. External Variability

During the SPLE lifecycle, variability is an entity introduced from the development of the product management and refined from requirements engineering until design and realization processes. Starting from requirements engineering, the stackeholders’ needs, laws and standards define all the variabilities. Later on, during the different processes of domain engineering, additional variability might be defined from the specific needs of internal development. For this purpose, we need to define a first categorization of variability: *external* and *internal* variability.

External variability refers to the possibility of customers to choose among different variants. Contrarily, the internal variability is transparent to the customers since it belongs to the needs of specific implementations or marketing decisions of the involved software company. Specific implementation decisions could be represented by different programming languages adopted for different purposes. In other cases, the introduction of internal variability could be caused by certain specifications that must be kept undisclosed in view of marketing decisions. The reason might lies on the fact that having too many specifications of technical choices could represent a de-motivational factor for user-experiences. In some other cases not all the product features are disclosed to the public information in view of competitors-related issues.

The reader should notice that the source of internal variability are different: introducing different programming languages might be a choice driven by the needs of a defined external variability such as different computational performance or might be due to organization constraints in view of marketing decisions. Therefore, on one hand the more the level of sub-processes goes toward the implementation, the more external variability vanish as the concern of the final costumers. On the other hand, internal variability increases in prominence as fact of transparency of detailed refinement decisions.

The following figure reveals the variability pyramid to graphically describe how the (internal and external) variability is introduced and related to the different sub-processes within domain engineering.

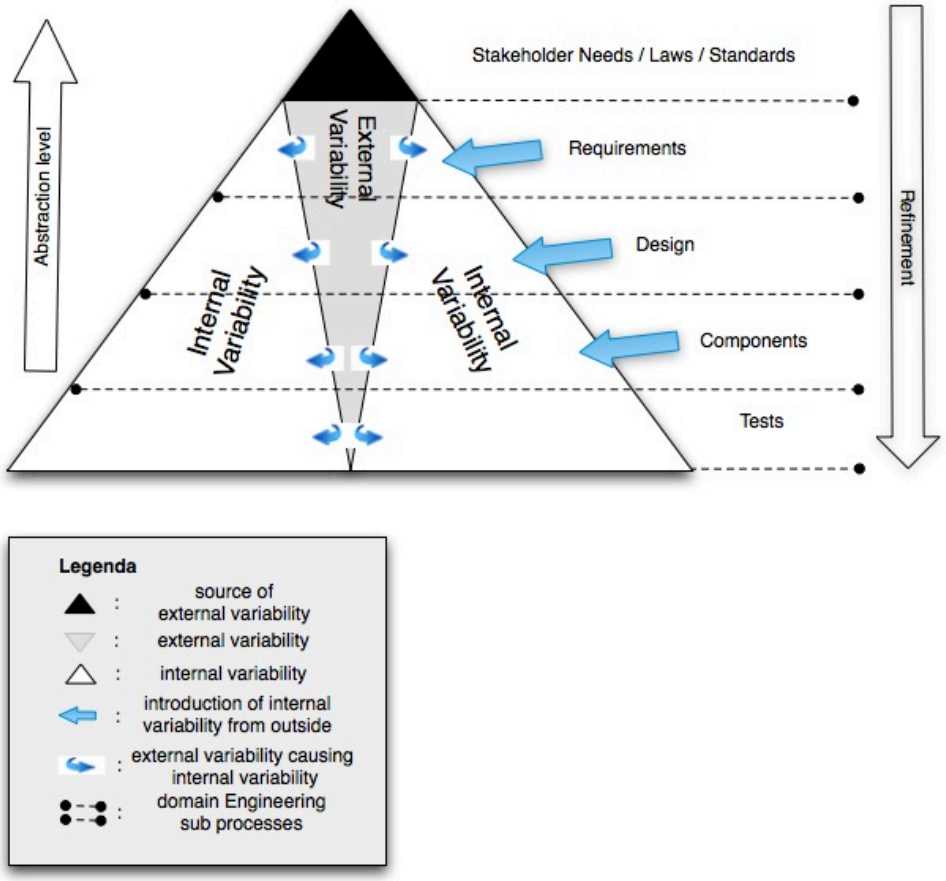


Figure 3.1: Variability pyramid [3]

After the variability is designed in the Domain Engineering, it is exploited in the Application Engineering for differentiating the several products. In order to facilitate the configuration of different products, all the artifacts that must be designed and developed with focus on flexibility. For instance, during design engineering, flexibility concerns to the potentiality of changing specific components, without affecting the entire architecture. This is called *variability in space*. The variability can manifest itself also in different times. In this case, the SPLE artifacts must be developed with evolvability concerns, where new entities (requirements, components etc.) should be easily introduced.

The introduction to the role of variability is crucial in our case study for supporting several activities like: variability definition; managing variable artifacts; supporting activities for resolving variability; collecting, storing and managing trace information necessary to fulfill these tasks.

The approach we described provides a base-line for the customization of different products, but in order to be effectively used it needs an artifact model, which describes how the product line artifacts are built and instantiated. The orthogonal variability meta-model (OVM) is introduced to standardize the representation of the variability models among the several artifacts [8]. The variability model will

be constructed by expressing the relation of variation points and variants with the use of constraint, variability and artifact dependencies as shown in the figure below.

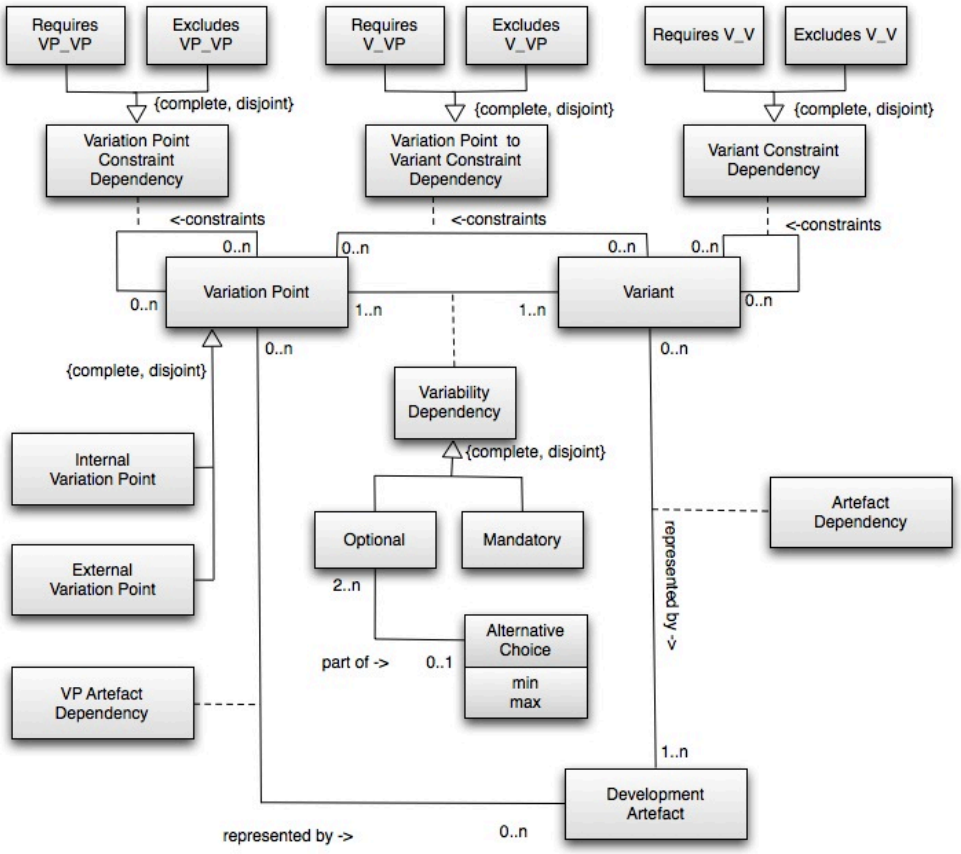


Figure 3.2: Orthogonal variability meta-model [3]

OVM is only one of the existing models for expressing variability. The difference between this model and others, as feature model could represent, is that OVM focuses on variability only, instead of documenting either common aspects of a product-line [3]. At this regard, since the commonalities of our case study are too many and tedious to represent, the focus will target only the variability dependencies and constraints configured by OVM.
An example of a final model is shown in the following figure.

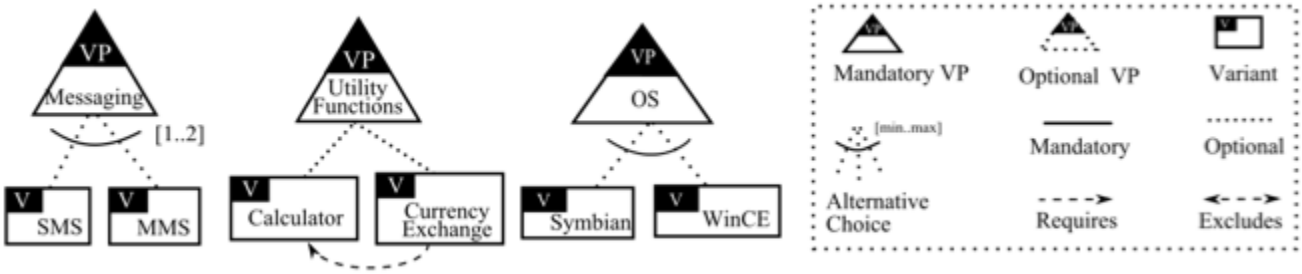


Figure 3.3: Feature Model to OVM Transformations [11]

The first variability presents how a messaging variation point might have from one to two variants through the alternative choice between SMS and MMS. In the second, we can notice the addition of a *requires v_v* dependency (see *Fig. 3.2*) between the two variants, indicating that no currency exchange can be selected if the calculator is not in place. In the third example, the operating system must have an alternative choice that is mutually exclusive (*Symbian* and *WinCe* cannot occur in the

same product), that is expressed by the cardinality [1...1] (omitted by default) in the notation. For more understanding of the use of this model, other applications will be presented in detail during the domain requirements section.

3.3 Product-Lines vs. traditional approaches

The definition and management of the variability differentiates the concept of SPLE from both the single system development and software reuse. The difference from single system development lies on the definition of the *variation point*. Single system development provides the implementation of a specific product without defined variability. When a product let the users to change a defined feature, it is considered as a *commonality* since it is shared among the different products. Variation points can only be defined by the customers before the installation phase of the output product.

To accomplish this, SPLE introduces the variation points through *domain engineering artifacts* (*i.e.* the product-line) and not through the application sub-processes (*i.e.* the products), where the variants are already specified. Examples can be referred to configuration management activities, where product features are defined before the product/service is installed: a premium edition might represent a variant for accessing to additional features respect to the normal edition of a specific product.

The difference from *software reuse* and SPLE, is that the concept of reuse in SPLE is planned, enabled and enforced. The domain engineering artifacts are designed to be reused and optimized in order to use them multiple systems. If we think software reuse as taking developed components from a previous work, for instance, we create a new product, which leads to different maintenance activities. Instead, if we think to SPLE approach, we reuse a common and managed set of features explicitly designed for reuse and we maintain the product line as a whole, not as multiple products. This characteristic role is prominent in view of economical benefits in production : using applicable components from the base of common assets and tailoring them in according to pre-planned variation mechanisms is more convenient compared to other strategies.

4. Domain Engineering in Practice

Domain Engineering is the set of sub-processes that focus on the construction and maintenance of the product-line platform. Variability and commonality are identified and realized throughout the process extracted from the *Fig 2.1* and shown in the picture below.

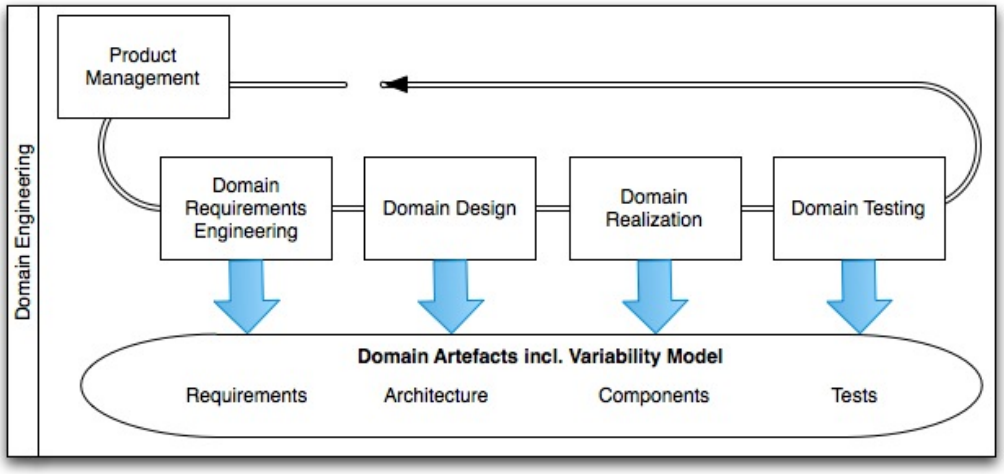


Figure 4.1 : Domain Engineering sub-processes overview

Building reusable artifacts, defining and refining commonalities and variabilities documented in the orthogonal variability model and realizing the variabilities are the main topics of this section. So, in the remaining of this section each sub-process will be analyzed and discussed in detail and where possible, a practical example related to the project we realized.

4.1 Product Management

According to Pohl's definition , *"Product management is the sub-process of domain engineering for controlling the development, production, and marketing of the software product-line and its applications"* [3]. The definition of the term *product* is related to goods, services or solution that are delivered to the market. The main output of Product Management is product roadmap which contains defining market and product strategy, common and variable features of products in addition to time schedule for products developments. Product Management has some relation with other sub-processes of Domain Engineering. In the next Figure we will show the relation among them.

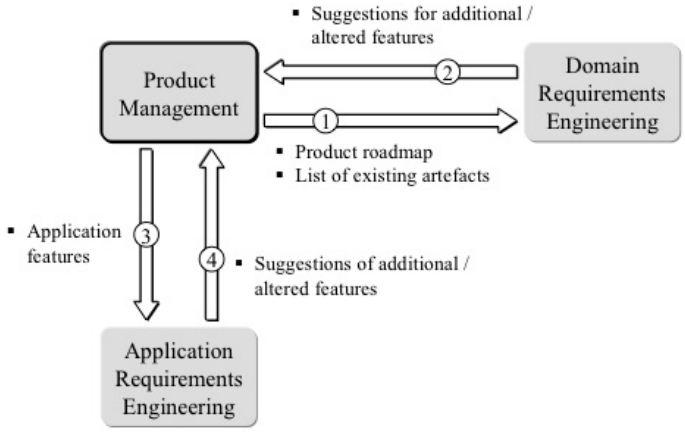


Figure 4.2: Information flow between product management and other sub-processes

Product Management in *Step 1* sends the product road-map to the to domain requirements engineering which shows the initial products in the product-line with commonalities and variability in addition to

any existing artifact from former projects in the organization. Some times, the similar projects have been implemented and its artifacts are available which can be reused in this project as well. In *Step 2*, Product Management is informed about any suggested changes in product and its features from Domain Requirement Engineering. Another output from product management is the *Step 3,* which application features are sending to the "Application Requirement Engineering" in order to analyze the products' requirements. *Step 4* shows the suggestions and feedback from Application Requirement Engineering to be applied in the Product Management.

Product Management Activities

//PM Activity 1 : Portfolio Management

One of the most important part of product management is *portfolio management*. The portfolio management focuses on the strategic aspect of portfolio and then designing new product or improve the existing products. In other words, portfolio management is about making decisions for existing or planning products or business in a company, which will monitor the different dependencies with other products or businesses. In order to reach the market, fulfilling the customer's requirements is a must. In this context processes and new projects are evaluated and suitable features are selected and prioritized. The main activities involved in this phase are as follows:

- Market introduction and observation: specify the market strategy and observe the current market
- Specify the business model: identify the type of business, which determines the product business and system business.
- Specify the product life-cycle: identify the different phases for the product : growth, maturity, saturation and degeneration. The platform life-cycle can also be discussed.
- Specify the interdependencies with other products

//PM Activity 2 : Product Definition

When a new product is introduced, the strategic gap within the product portfolio has to be covered : functional and non-functional features are specified to fulfill the customer's demands. To find features of products different methods, techniques and suggestion are released. As an example, Kano introduced a method for selecting the proper product's features [10] by classifying the requirements into different categories : *basic*, *satisfier*, *delighter* and *indifferent*. The *Kano method* also helps to determine the common and variable features.

Sometimes successful products are already developed and maintained by other companies, which are similar to the product that is being developed. In this case, the strategy called *imitation strategy* has to be applied in order to overcome the market by pushing innovation into the new product. In this context, it is crucial to take into account key capabilities of the product such as technology, marketing and production.

//PM Activity 3 : Managing existing products

Some times managing available products in companies is the case. Maintenance of existing products can be divided into *corrective* (fix bugs), *adaptive* (fit to the changing environment), *perfective* (improve the system) and *preventive* (prevent malfunctioning). Product-lines should be designed in such a way that makes it easier for developers and designers to add and modify features and make the

platform evolvable. Each new deletion/addition/modification to an existing product should be made explicit and recorded to a central system to make it easy to track and monitor.

//PM Activity 4 : Scoping

In software product line engineering, scoping consists in identifying the products that will be part of the product-line and defining the product-features. We can divide activities of scoping in three groups :

- Product portfolio scoping: defining products and their key features
- Domain scoping: defining boundaries of the domain
- Asset scoping: Define components that can be designed for re-use

//PM Activity 5 : Output

The product road-map is the main output of product management which is used by domain engineering to provide the requirement list. The product road-map contains common and variable feature list for all products of product line and schedule plan for developing the products or services. This road-map is used by domain and application engineering to provide related artifacts by using its feature list. Also list of the other products of organization that can be used in the product line are included in the output.

Example : Product Management

In this section, the product management will be explained in the form of a product-line example titled *Web-line.* It presents product definitions, scopes and output separately. The final output of this section is the product line road-map that is delivered to domain engineering and application engineering for further use. As there are no similar products in the organization for the sample product line, the activity for managing exiting product in this section is missed.

// Product Definitions

The *Web-line* generates two type of product, which are new to the organization:

- P.1 : Personal website
- P.2 : Company website.

In a personal website, persons introduce themselves, their portfolio and contact information while in a company website the history, product and services in addition to available job positions, contact information will be introduced.

//Scoping

The website itself for persons (P1) and companies (P2) can be divided into different pages. The "*Homepage*" and "*About*" are available for both the product types. The product for personal usage will include "Portfolio" web-pages while company's product has "Product and Services" and "Job opportunity" web-pages. Although both application type has "Contact" web page but this page will show map of the company in the case that the company is the customer. Each page is made of a structure and a style that combined together with header, menu, content, side-bar and footer forms what we defined as a *template.*

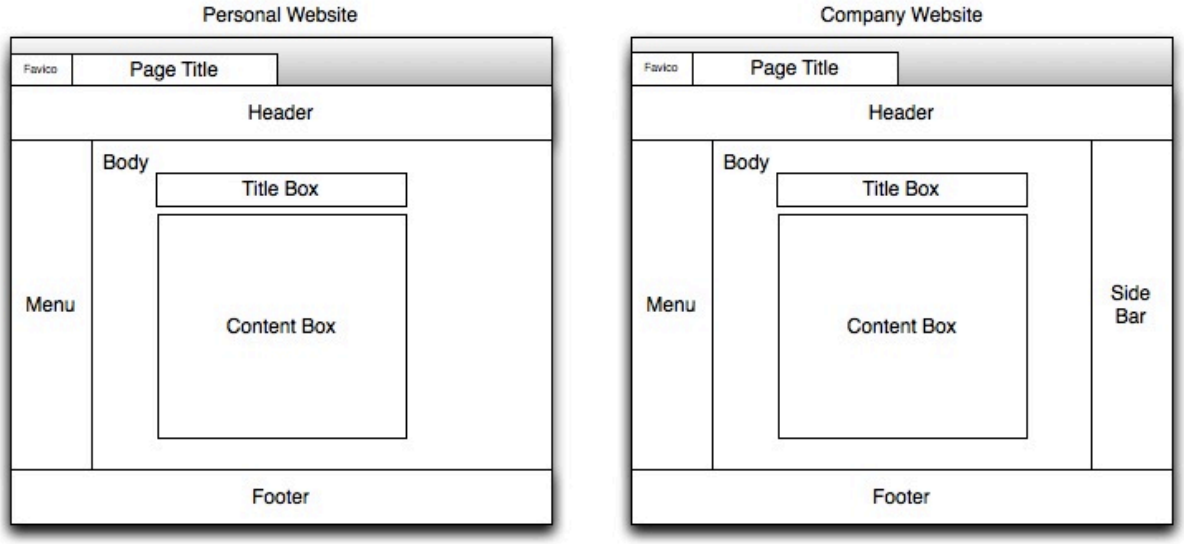


Figure 4.3 : Template structure

Content itself has an internal layout as well, which shows the arrangement of internal elements such Title, body text and images that can vary among different pages. While this structure is fixed for all the pages of the product, the layout of what is inside the *Content Box* changes for each page. In the remaining of this section we will present the layout of each pages that will be inserted inside the *Content Box* of the *Figure* above.

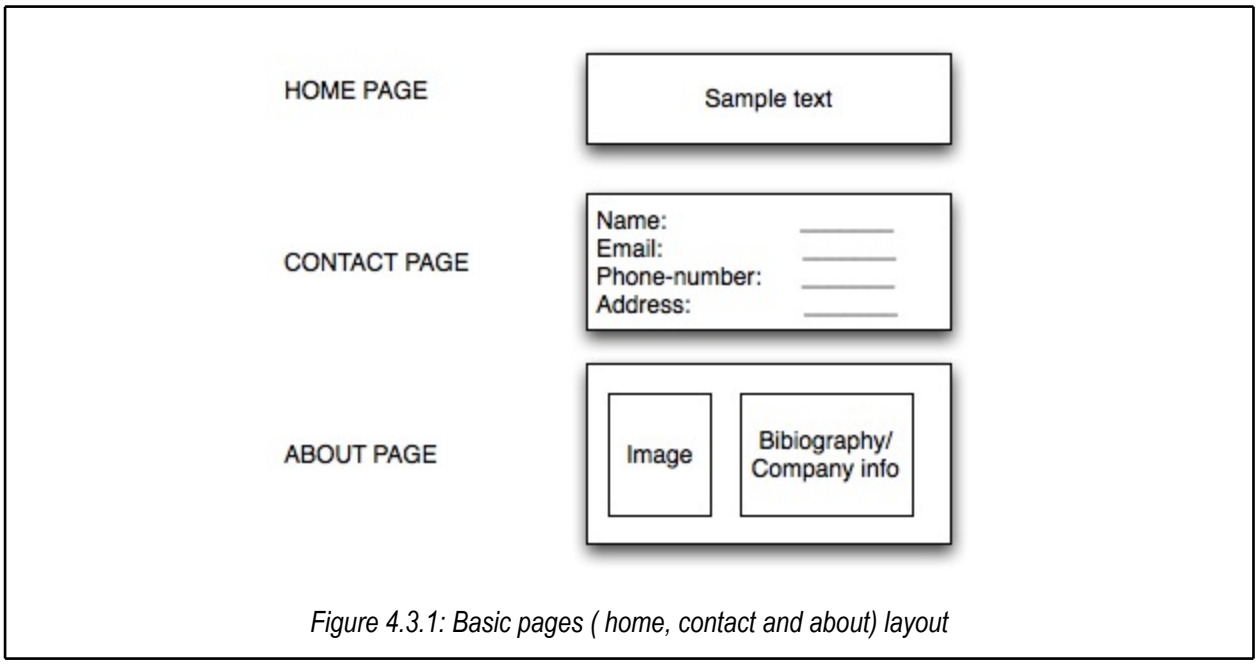


Figure 4.3.1: Basic pages (home, contact and about) layout

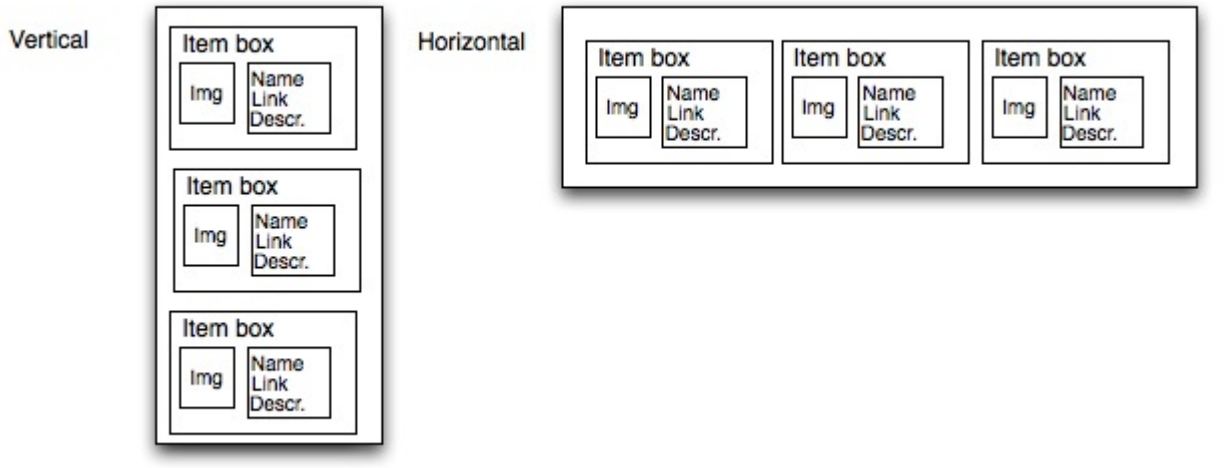


Figure 4.3.2: Portfolio page structural elements [in two configurations]

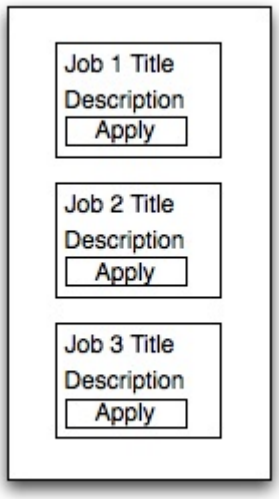


Figure 4.3.3 : Job Opportunity page structural elements

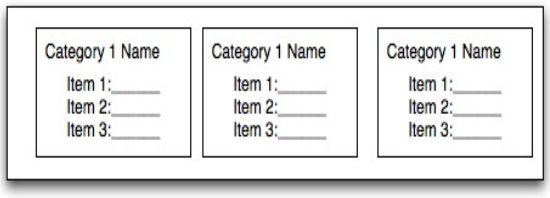


Figure 4.3.4 : Product sand Services page structural elements

HTML and CSS are the fundamental assets that are available for building web-pages. Web pages should be displayed through a web browser, which supports HTML 4.0.

In defining product scoping, the key features and functions of product line and its products are defined. The following functions are involved when related assets are available:

- Creating web pages accessible from a browser
- Organize contents of the page in sections
- Separating *html* from *javascript* and *CSS* files (MVC)
- Selecting among different background colors
- Organize subsections in pages
- Visualizing sub-pages in a menu

- Creating a footer containing social icons
- Creating a page to show contact information
- Create the about page
- Creating an home page
- Choose between optional sub-pages
- Integrating a system for creating visitor statistics
- Using social networks buttons for let user *appreciate* the website
- Link to professional information (Linkedin)
- Changing the title of each page
- Customize the *favicon* of the website
- Customize the *header* of the website
- Implement *Jquery* library in every page

//Output

This part clarifies the output of the product management. General outputs of the product management should be a road-map that identifies the products features and time schedule for developing the products. So for defining the product line road-map, the output of the product-line is also included:
The outputs of product-line are two families of product: Personal websites and Company websites. They are composed by static files such as HTML, CSS , javascript and images that are delivered (zipped) to the Webmaster (our *customer*) to be deployed in a web-server and be finally utilized by *end-users*. The features offered by the *Web-line* could be summarized as:

- A *platform structure* containing basic settings and layout;
- Default web-pages: Home, Contact and About;
- Additional web-pages in according to the type of the product (Personal or Company);
- Deploy products on any modern browsers that support at least HTML 4.

4.2 Domain Requirements Engineering

Introduction

Domain requirement engineering identifies the requirements that are common between all products and the requirements, which are product-specific. In fact during this sub-process, commonality and variability analysis should be conducted in order to generate the variability model that will be explained in the next sections. Traditional requirement engineering is conducted in elicitation, documentation, negotiation, validation-verification and management processes. But according to Pohl ,it can be categorized in three dimensions: *understanding requirements*, *representation of requirements* and *agreement about requirements* [23].

Requirements are usually documented by natural language or requirement modeling which serve an input to the design phase of software development. The first question is where the requirements come from. When we define requirements in SPLE, we have to pay attention to the neat distinction between product requirements and product-line requirements. Each typology of requirement have a different group of stakeholders, depending on the specific situation.

Requirements are documented by using the natural language combined with other specific tools like

use cases, virtual windows and so forth. Therefore in each model, it is possible to find variability of the requirements and map it to the variability model.

The following activities should be fulfilled in requirement domain engineering:

- Commonality analysis
- Variability analysis
- Variability modeling
- Document requirements in a suitable model
- Provide requirements artifact (scenarios, use cases, etc.)

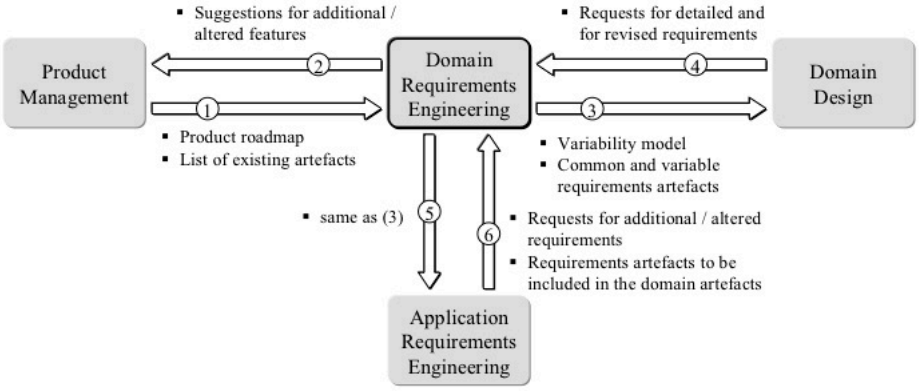


Figure 4.4: Information Flows between Domain Requirement Engineering and other sub-processes

The *Figure 4.4* presents the information flows between Domain Requirement Engineering and other sub-processes of the product line. Product Roadmap in addition to existing artifacts from previous projects are forwarded to Domain Requirement Engineering (*Step1*). During requirement engineering, some times changing product's features may be required. In *Step 2*, when a feature is added or changed in the Domain Requirement Engineering, the Product Management is informed about this change. Variability model accompanied by commonality and variability of Domain Requirement Engineering is passed to the domain design phase (*Step 3*) and a request for revising requirements are forwarded back from Domain Design (Step4). Domain Requirements Engineering also provides variability model in addition to artifacts of common and variable requirements for Application Requirement Engineering, in order to use them in requirement engineering phase of a special product(Step 5). When requirement engineering is conducted in Application Requirement Engineering, a request for additional requirements or new requirement artificats may be sent back to Domain Requirement Engineering from Application Requirement Engineering (*Step 6*).

Commonality Analysis

While conducting the requirement elicitation, list of common requirements among products has to be identified. Commonality analysis is performed to have as much commonalities as possible between products, and thereby to reduce the amount of variability to the minimum [4]. In other word, the same requirements that are mandatory for all products are identified as the common requirements.

In order to provide a list of common requirements, an *application-requirements matrix* is prepared to extract the common requirements out of all products' requirements. This matrix lists all available

requirements including common and variable features of road-map in the left column and different available applications of product-line at top row. Mapping between requirements and products presents a matrix. The hit indicates product has met the related requirement.

Similar to providing common requirements, common goals are also specified by identifying the goals of each application and finding similar goals between them. Therefore, each product of the product line is defined through goals, desired features and requirements specification. Goals represent the intentions of each customer and/or market segments to be achieved. Desired features represent what the specific product can achieve through the encompassed components.

Requirements specifications list all the functionalities of the products in detail. Some requirements may not deal with functionality and are discussed as the non-functional requirements, which are also mentioned in the requirement list. In order to avoid replication of the common requirements in different products, a separate list of requirements is provided containing common functional requirements.

Example : Commonality Analysis

The main goal of this example is presenting how common requirements are extracted out of all products requirements in the *Web-line* example. So, initially *application-requirements matrix* is provided which is presented in the *Table 4.5*. In the matrix, we try to map different types of application with all requirements. So if one requirement is available on all application (such as *R1* and *R2* in the *Table 4.5*, it is tagged as a common requirement. This table presents a model to extract all common requirements. So it is obvious that in the case with too many products for one product line, providing such table is time consuming and is not a recommended model.

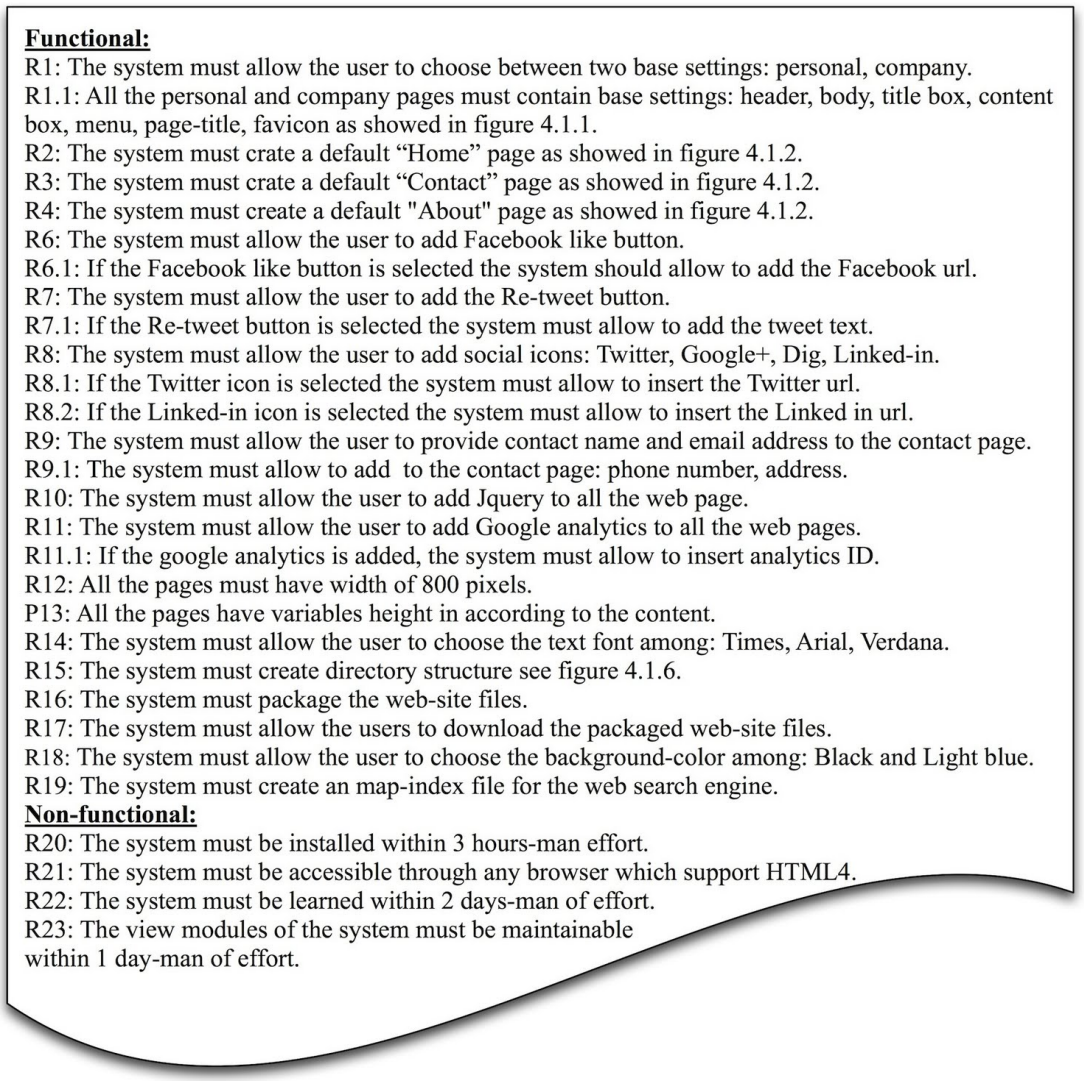
Application Requirement	App1 (A personal website)	App2 (A personal website)	App3 (A company website)
R1	✓	✓	✓
R2	✓	✓	✓
. . .			
R25	✓		
R30			✓

Table 4.5: Application Requirement Matrix

According to the explained features of *Web-line* example as the output of Product Management toward the Domain Requirement Engineering, the common goals for Web-line similar to common requirements are identified as follow:

- Achieve a good level of online presence
- Save time and costs to produce the website
- Make information available worldwide 24/7
- Keep communication with social entities
- Enlarge the pool of user that access the information

Afterwards we identified and prepared a list of *common* required which are shared both from *P1* and *P2*, as listed in the figure below.



Functional:
R1: The system must allow the user to choose between two base settings: personal, company.
R1.1: All the personal and company pages must contain base settings: header, body, title box, content box, menu, page-title, favicon as showed in figure 4.1.1.
R2: The system must crate a default “Home” page as showed in figure 4.1.2.
R3: The system must crate a default “Contact” page as showed in figure 4.1.2.
R4: The system must create a default "About" page as showed in figure 4.1.2.
R6: The system must allow the user to add Facebook like button.
R6.1: If the Facebook like button is selected the system should allow to add the Facebook url.
R7: The system must allow the user to add the Re-tweet button.
R7.1: If the Re-tweet button is selected the system must allow to add the tweet text.
R8: The system must allow the user to add social icons: Twitter, Google+, Dig, Linked-in.
R8.1: If the Twitter icon is selected the system must allow to insert the Twitter url.
R8.2: If the Linked-in icon is selected the system must allow to insert the Linked in url.
R9: The system must allow the user to provide contact name and email address to the contact page.
R9.1: The system must allow to add to the contact page: phone number, address.
R10: The system must allow the user to add Jquery to all the web page.
R11: The system must allow the user to add Google analytics to all the web pages.
R11.1: If the google analytics is added, the system must allow to insert analytics ID.
R12: All the pages must have width of 800 pixels.
P13: All the pages have variables height in according to the content.
R14: The system must allow the user to choose the text font among: Times, Arial, Verdana.
R15: The system must create directory structure see figure 4.1.6.
R16: The system must package the web-site files.
R17: The system must allow the users to download the packaged web-site files.
R18: The system must allow the user to choose the background-color among: Black and Light blue.
R19: The system must create an map-index file for the web search engine.
Non-functional:
R20: The system must be installed within 3 hours-man effort.
R21: The system must be accessible through any browser which support HTML4.
R22: The system must be learned within 2 days-man of effort.
R23: The view modules of the system must be maintainable within 1 day-man of effort.

Figure 4.6 : Common Requirements

Then we proceed into identifying requirements that are product-specific. In the next figures we show the requirement specifications of the personal website (*P1*) and company website (*P2*).

Goals
Introduce personal skills and abilities
Find suitable jobs in according to personal skills
Win competition of selecting jobs
Keep communication with companies
Desired Features
create “Home” page layout
create “About” page layout
create “Contact” page layout
create "Portfolio" page layout
link to social networks (Professional, not professional)
Requirements
(in addition to the common requirements)
R24: The system should allow the user to choose Pink as additional background color to Black and Light blue
R25: The system must create a default "Portfolio" page as shown in figure 4.1.3.
R26: The system must present the personal web-pages structure as presented in figure A 1.1.

Figure 4.7 : Personal website Requirements

Goals
Increase the possibility of taking the empty job position.
Propounding the company.
Increase the visibility of the products and services the company offers.
Desired Features
present Organizational map.
create “Product & Service” page layout.
create “job opportunity” page layout .
showing news from the Company Facebook page.
showing in-line news from one RSS source.
Requirements
(in addition to the common requirements)
R27: The system must allow to add the map in the "Contact" page as shown in figure 4.1.2.
R28: The system must create a default "Products & Services" page as shown in figure 4.1.5.
R28.1: The system must allow to add from 1 to 3 categories in Product & Services.
R29: The system must create a default "Job opportunity" page as shown in figure 4.1.4.
R29.1: The system must allow to add the job-email address in Job Opportunity page.
R30: The system must allow the user to add Facebook social plug-in.
R31: The system must add the side-bar in basic settings.
R31.1: The system must add the contact box in the side-bar containing name of the user and email-address.
R31.2: The system must allow to add to the side-bar additional front-end components: News-ticker, Facebook social plug-in
R31.2.1: If the Facebook social plug-in is selected the system should allow to add the Facebook url page.
R31.2.2: If the news ticker is added, the system should allow to insert url of RSS feed.
R32: The system must present company web-pages structure as presented in figure 4.1.1.

Figure 4.8 : Company website Requirements

Variability Analysis

Providing the variability model for the product-line is the challenging part of Domain Requirement Engineering. Variability analysis identifies the requirements variability and defines the variation points and their variants related to the specified requirements. The application–requirements matrix that is exampled in *Table 4.5* , is also used to address variables among requirements of different applications. In other words, if one requirement is not available in all applications, it means that there is variability point with related variants and therefore, the requirement is mapped to one of those variants.

When all variation points, variants and their relationship are put together in a form a model, the generated model is called variability model. In order to increase the clarity of the variability model, a good practice is to map each requirement to relevant variants. It can be done simply by writing the requirement's identifier beside the related variant.

We can summarize the phases in variability analysis as follows:

- Identify the variable requirements
- Determine the variation points and variants
- Define the variability and constraint dependencies
- Map variants to requirements.

Example : Variability Analysis

In this example we present the orthogonal variability model of *Web-line* through a set of figures. The *Fig. 4.9* shows the available web-pages and their layouts for the personal websites and company websites.

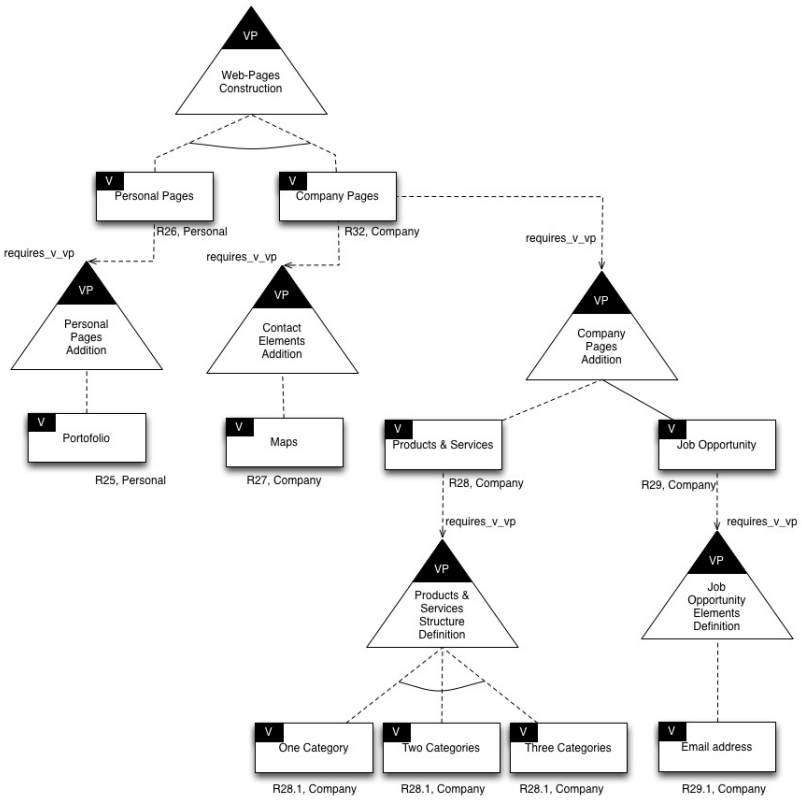


Figure 4.9: Web-pages variability model

All the variants will be associated to the requirements described in the section above. An exception to this is the requirement *R19,* which involves the implementation of an *internal variation point* that is *SiteIndex.* Both P1 and P2 will have a *sitemap.xml* markup file [5], but it will differ in terms of specific attributes that are more specific for companies. Personal web-site will reach low frequency of page-scan and same priorities for all the pages since it rarely tends to change, whilst Company web-site will reach high frequency of page-scan and highest priority of default pages respect to the additional pages in view of the marketing competition with other companies on the web (details will be presented in the lower level design decisions). As this difference will be transparent for the customer this is defined as an *internal variability.*

The next variability model (*Fig. 4.10*) shows the background color for both company website and personal website in addition to components for company website. It should be mentioned that in order to make the model clearer, the "*web-page construction*" variability point has been repeated in this model as well as shown in the figure below:

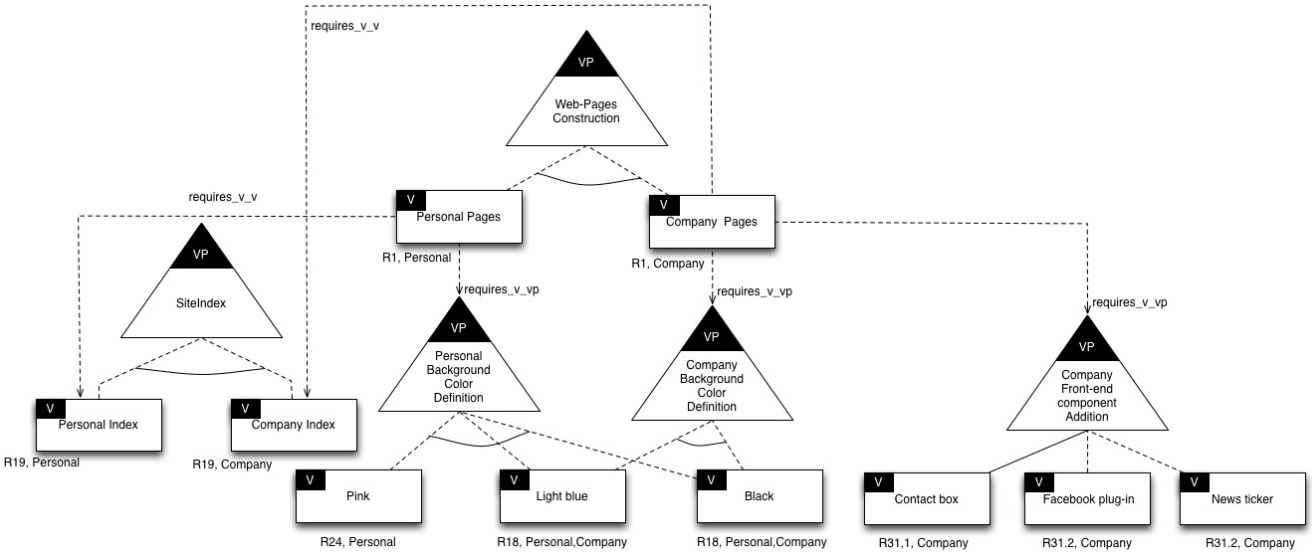


Figure 4.10: Components and background-color variability model

Modeling variability requirements

As we mentioned above, requirements can be presented by natural language or a requirement representation model such as use-case diagram. When requirements are modeled with natural language, the mapping between variants and requirements can be done by introducing requirement identifiers in the variability model.

Also a use-case diagram can present the system requirements in a form a graphic overview of the functionality, actors and the relationships between use-cases and actors. Each use-case is utilized for documenting functional requirements, which contains a scenario to show how it should interact with stakeholders or other cases. In order to analyze the variability model better, the variants should be specified in the use-cases.

Example : Modeling Variability Requirements

The *Fig. 4.11* shows the model of variability requirement based on use-cases of the product-line, which functionalities are explained in the *scope*.

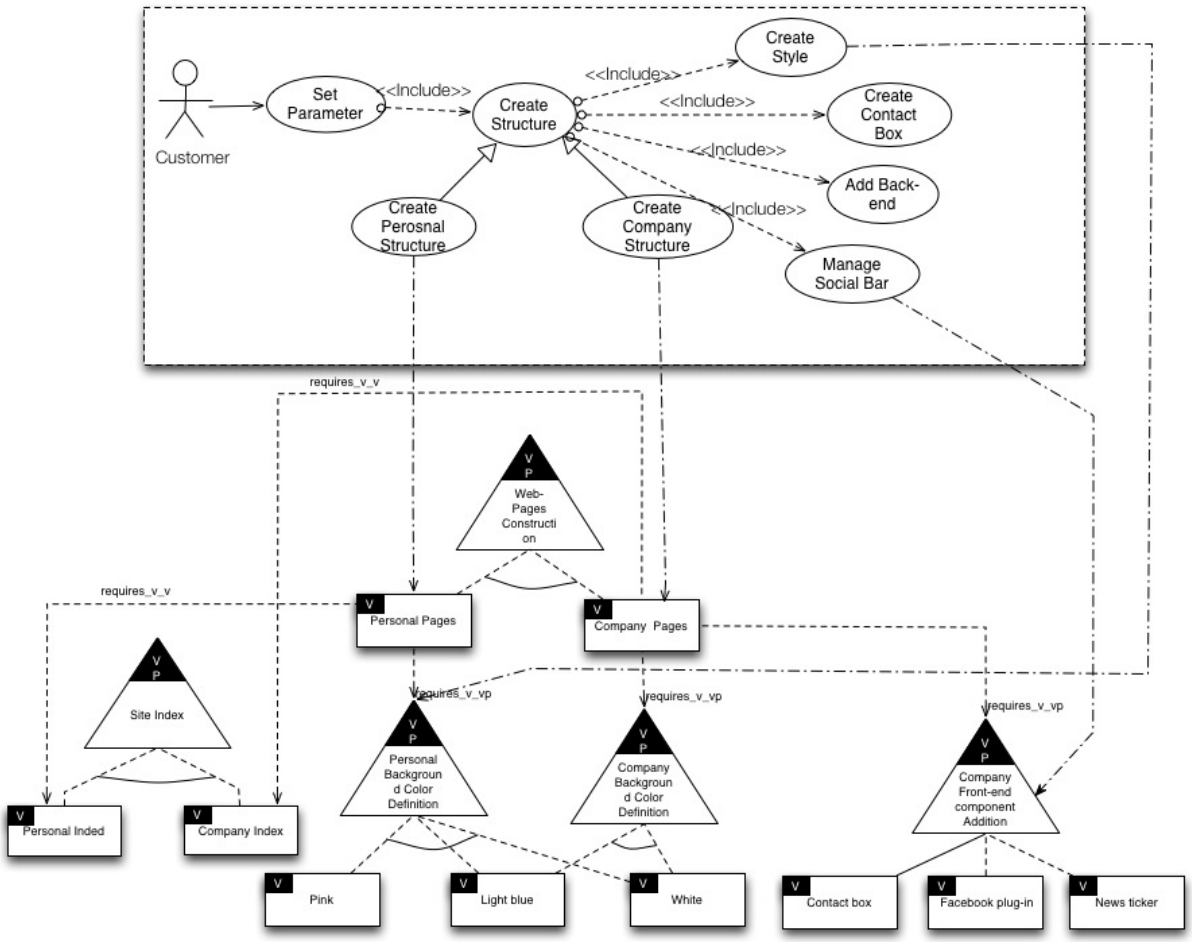


Figure 4.11: Variability model for structure and component use cases

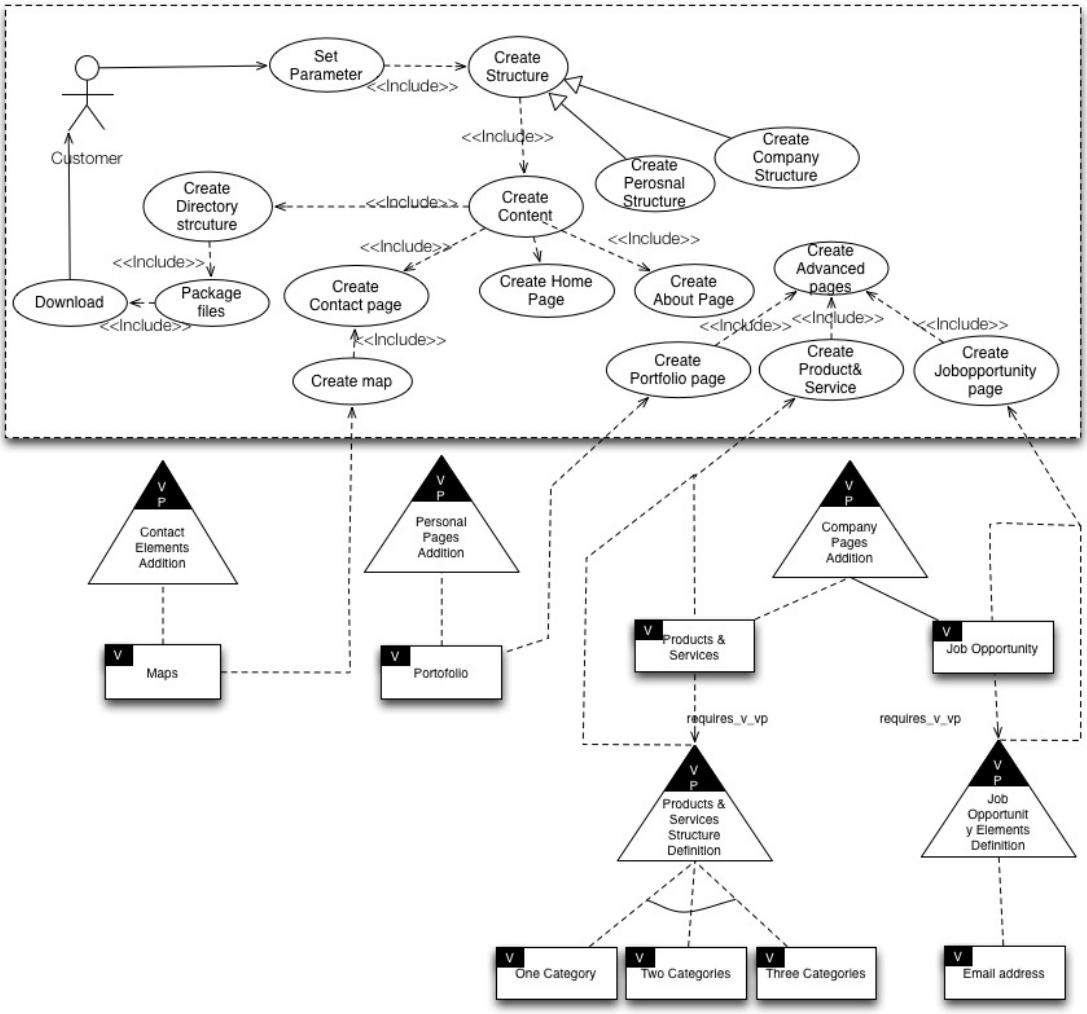


Figure 4.12: Variability model for web-pages use cases

4.4 Domain Design

The most important design activity is designing the system architecture, which specifies high-level structure of the platform. Requirements and their variability should be mapped to the suitable technical solutions to be implemented in the realization phase.

The Domain Requirement Engineering is closely related to Domain Design. As it is presented in *Figure 4.10,* variability model, common and variable requirement artifacts, which have been prepared in Domain Requirement Engineering are sent to Domain Design (*Step1*). While the architecture is being designed in Domain Design, some times some ambiguous requirements or more requirements details are required by Domain Design, a request to review the requirement is submitted to the Domain Requirement Engineering (*Step 2*). Domain Design is responsible for providing the reference architecture for product-line and sends this architecture including variable structure to the Domain Realization (*Step 3*). In *Step 4*, issues in realizing domain artifacts are sent back form Domain Realization to the Domain Design. Reusable domain artifacts and interfaces are sent (*Step 4*) to Domain Realization to be realized and implemented. In *Step 5,* Domain Design provides the reference architecture for the Application Design in order to design the reference architecture for single application.

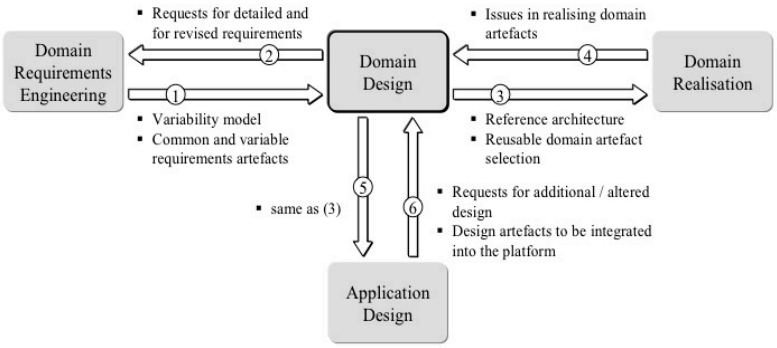


Figure 4.13: Information Flows between Domain Design and other sub-processes

Domain Design provides a structure for a set of products, which is referred as the product-line architecture and contains some generic components in addition to specific assets for each product. In this context, the individual product architecture is derived from the product-line architecture, which uses its own specific components.

In order to model the product-line architecture, artifacts from the traditional software architecture discipline can be used (e.g. Siemens' four-view model, Kruchten's 4+1 architectural view model). Hence, using the artifacts in architectural layers, component frameworks, modules and classes diagrams will present the architectural design.

Design the Architecture

The architect has to map the Domain Requirements to the technical solutions. The main objective of Domain Design is providing the a reference architecture that considers variation points. As we mentioned before, the reference architecture has to be flexible, evolvable, and maintainable. It should foresee some room for future incoming requirements and new technologies. The reference architecture generally consists several components, which are connected through interfaces [3].

Components and interfaces are important Domain Design artifacts. In designing a framework for architecture, common components and interfaces are extracted. For this framework, it is important to clarify it with more details by using other artifacts. So, other UML diagrams such as sequence diagrams, activity diagrams, collaboration diagrams, state diagrams and deployment diagrams may also be delivered. Although providing all mentioned diagrams make the architecture more clear, some times there is no need to deliver them all for the design as using number of them will be enough for clarification.

Commonality and variability in design

The majority of commonality and variability of design is originated from the analysis of requirements. In the next section we will see how the variability and commonality should be considered while creating the system architecture.

Mapping requirements on design

Requirements variability is an important source of variability in the reference architecture [3]. When a variation in requirements is present, the related variation in design realization should be consequently

considered. So it is important to map the requirements to the design artifacts.

In order to trace the relation between requirements and design, a traceability model is presented as shown in *Figure 4.11*. This model shows the relation between common and variable artifact, and variability points of requirement and reference architecture. It is used for tracing the map between requirement and design and also for tracking their relations in the future changes.

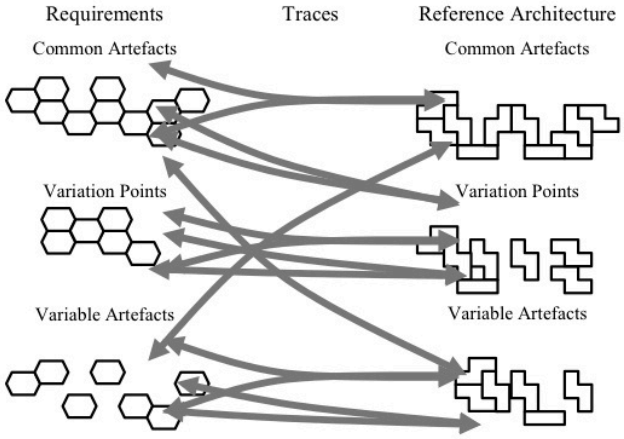


Figure 4.14 :Map between commonality and variability of the requirements and reference architecture

Add variability to the design with plug-ins

In the architecture design, it is important to specify the relation between design artifacts and variability diagram. There are several techniques that could be used in order to create a flexible and reusable architecture. Using specific application components and plug-ins in correspondence to the variation points are sample of making reusable architecture. The reference architecture identifies reusable assets in the software product-line but sometimes there are variants for a specific application. The application architect defines application-specific variant and consider a plug-in to be added to the reference architecture when is needed.

Reference architecture

The reference architecture contains components which are connected through their interfaces. A framework have common components and interfaces between products in addition to product specific components known as plugins. Using plugins helps the framework to be adoptable for further changes in future when new concern for adding new product specific pluging arise that may not affect the other part of framework.

As we mentioned before, in the case of product-lines, incorporating the requirements in the architecture by mapping them to the related components is the most important part of an architecture. To make it more clear, it is important to provide a model to show mapping between components and variability.

After component diagram is provided, the next activity is to prepare the module diagram and class diagram. Each component is mapped into one or more modules and then each module is mapped to classes and interfaces. This mapping can be conducted through a table.

Example : Reference Architecture

The *Figure 4.15* explains the different responsibilities of layers and sub-layers of the Web-line architecture. In the *View* layer, some options are presented to the costumer in order to let him customize the product preferences and obtain a set of customized parameters. Inside the *ServletMgr* layer, sub-layers have been configured to support a pipeline construction. From the *StructureMkr* to *ContentMkr* the product is created in two steps: first by configuring the general structure common to all pages (*StructureMkr),* and then introducing the content in each sub pages (*ContentMkr* and *AdvContentMkr*) . At the end *DSMgr* is responsible for creating a zip file, store it into the data-store and updating the view in order to provide the generated product to the customer.

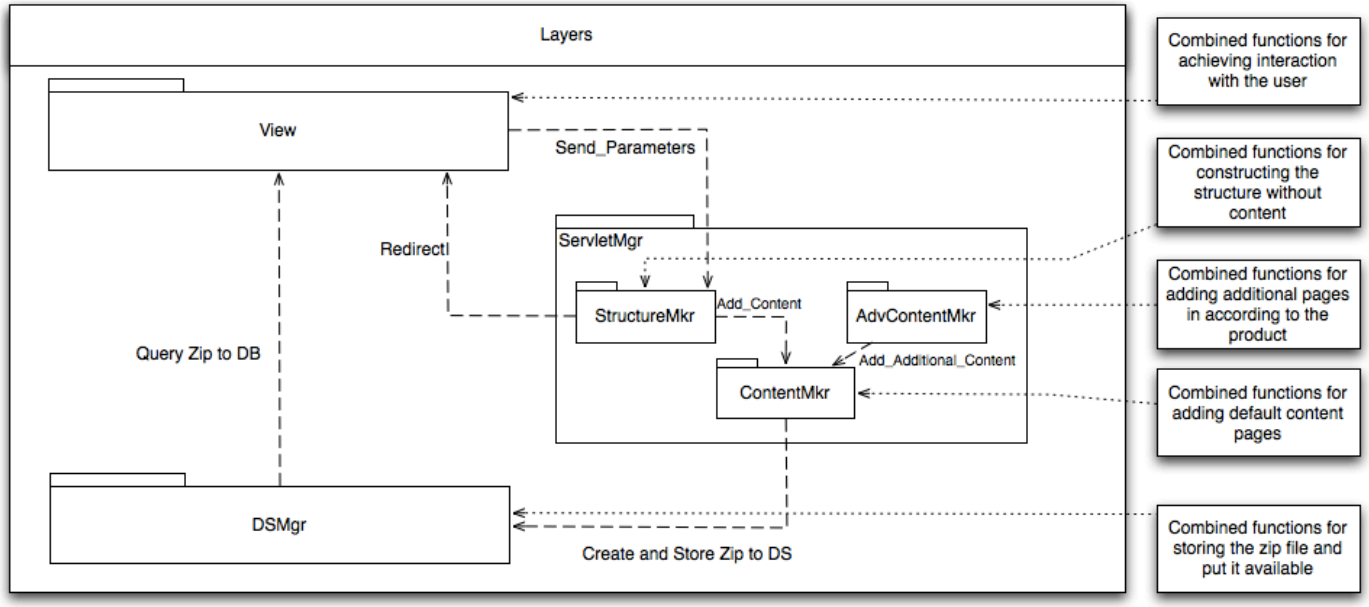


Figure 4.15: Layer Diagram

The Figure 4.16 explains how the components have been configured and related each other in according to the layer diagram previously discussed. To enhance evolvability of the pages that potentially could be added in *AdvContent*, the *BlackBoard* pattern style has been presented. Precisely, *BBController* is in charge for controlling plugins that are bound in the framework, whilst *BlackBoard* prepares for the integration of the adding pages results. This solution permit the architect to introduce new plugins for pages in the future, without changing the holistic structure. The StructureMgr is responsible component for preparing the structure of the webpage, which contains different boxes such as header, menu, footer and side bar boxes for company web site. It will create the holistic template of the product. ContentMgr component is in charge for integration of the multiple contents to the structure and provides html webpages. So template is designed and created in Domain Engineering as a common asset and submitted to the Application Engineering in order to facilitate creating new product specific webpages by just adding content to the pre-defined structure.

In addition of the default pages, it interacts with *AdvContent* in order to put extra pages in according to the requested parameters. At the end it interacts with *OutMgr* to send the final product, ready to be formatted for downloading.

It should be mentioned this diagram shows the updated design after getting feedback from Application Requirement Engineering to add new plugins called *Layout plugin*. This required change will be discussed later in Figure 5.3.

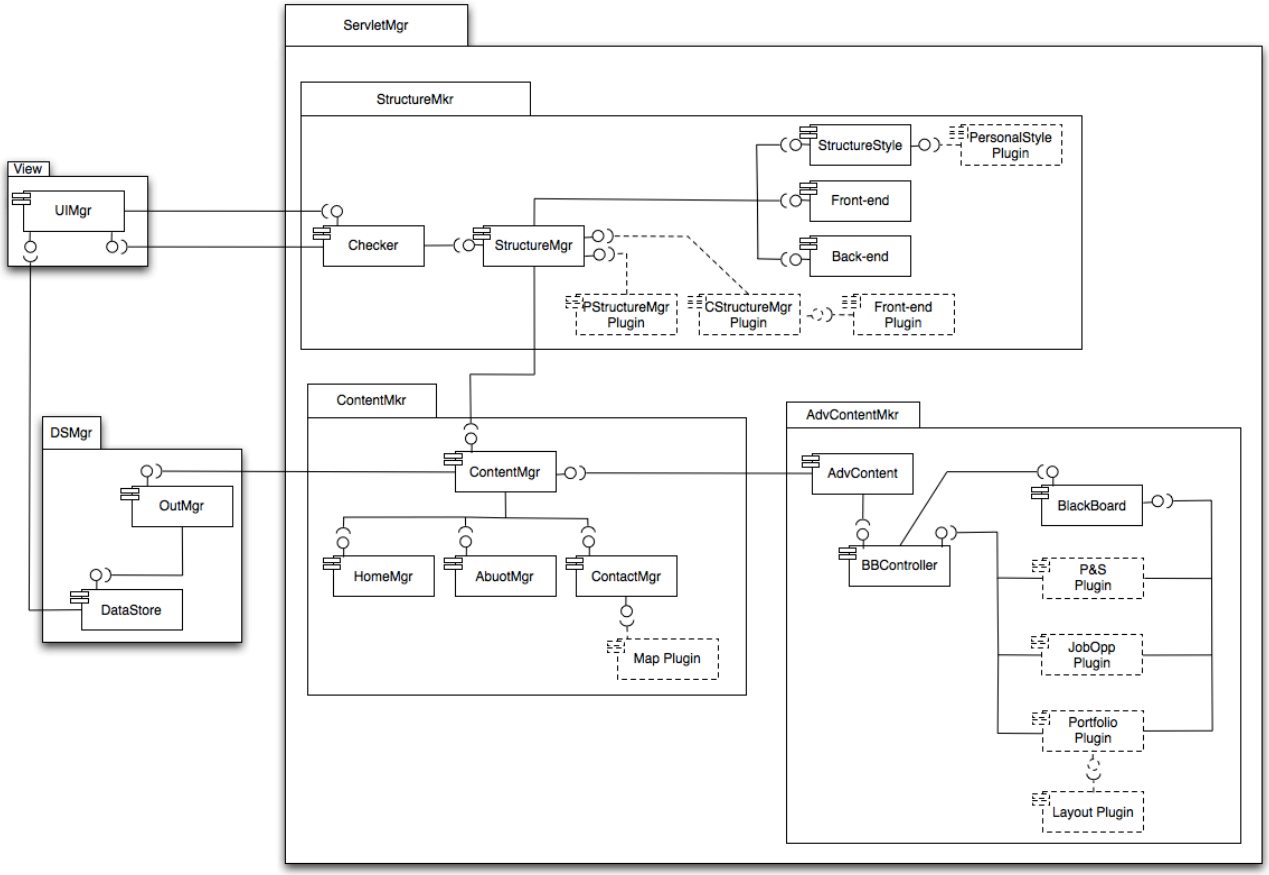


Figure 4.16: Component diagram

The rationale behind the creation of plug-ins can be found by reading the correlation to the variability model documented in the Domain Requirements Engineering. The following figures show the variability mapped to the component diagram.

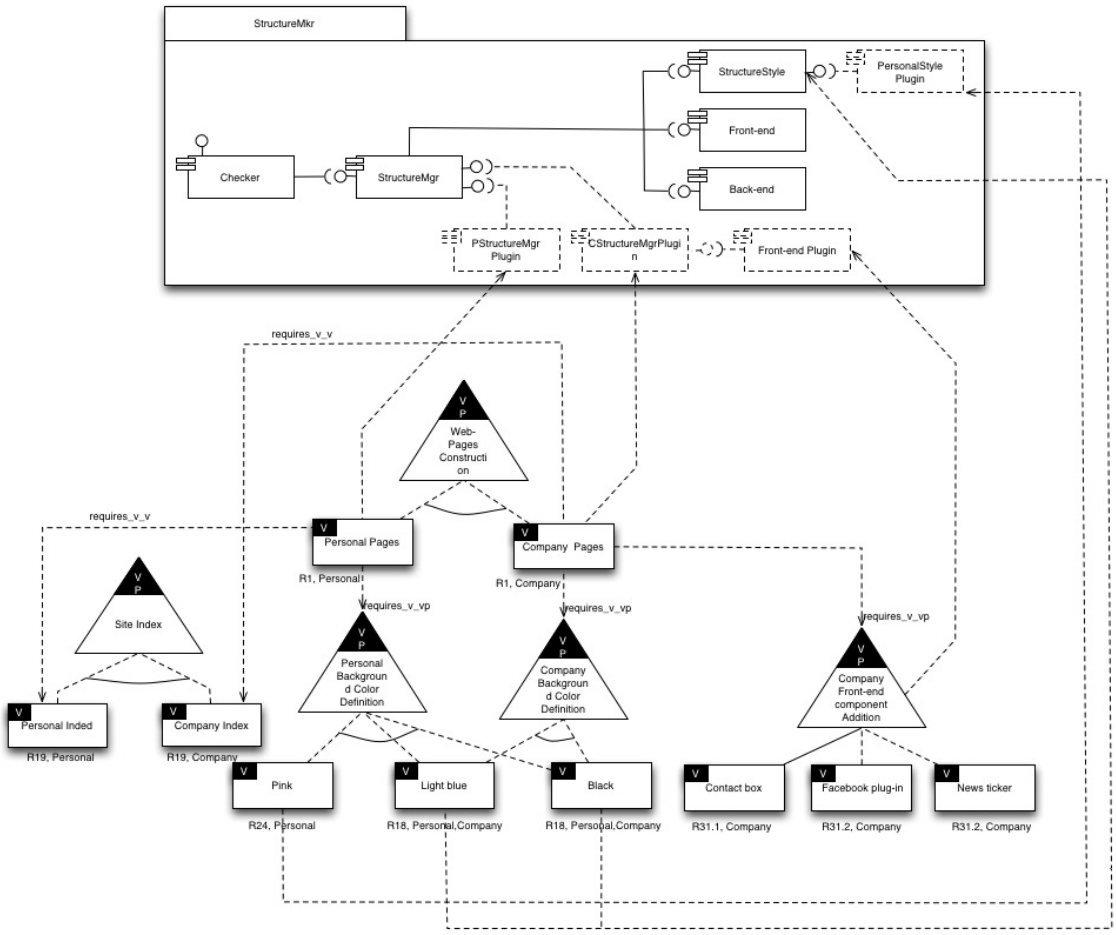


Figure 4.17: Variability in StructureMkr component

In *Fig. 4.17 StructureMkr* we introduce variability of general web-pages construction, style and company front-end as shown in the figure below. In addition *StructureMkr* contains internal variability of the *sitemap.xml* . *ContentMkr* involves variability of the contact-page elements. Maps can be added in case of a company web site is chosen as shown below.

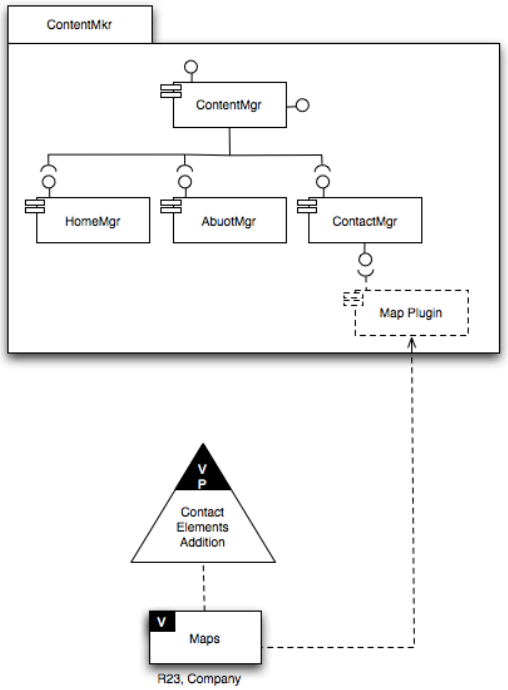


Figure 4.18: Variability in ContentMkr component

As it is presented in *Figure 4.19,* in the next page, *AdvContentMkr* contains variability of the additional pages specific for P1 and P2 (company and personal web site): *Products & Services* and *Job Opportunities* can be added for P2, *Portfolio* and its layout can be selected for P1.

As mentioned before, the variability of the pages has been treated as critical aspect in terms of structural evolvability of the platform. In this concern the *Black-Board* pattern has been introduced to be independent from the number of plug-ins that are installed. The addition of extra plug-ins will affect only configuration parameters where *BBController* component will read the available pages.

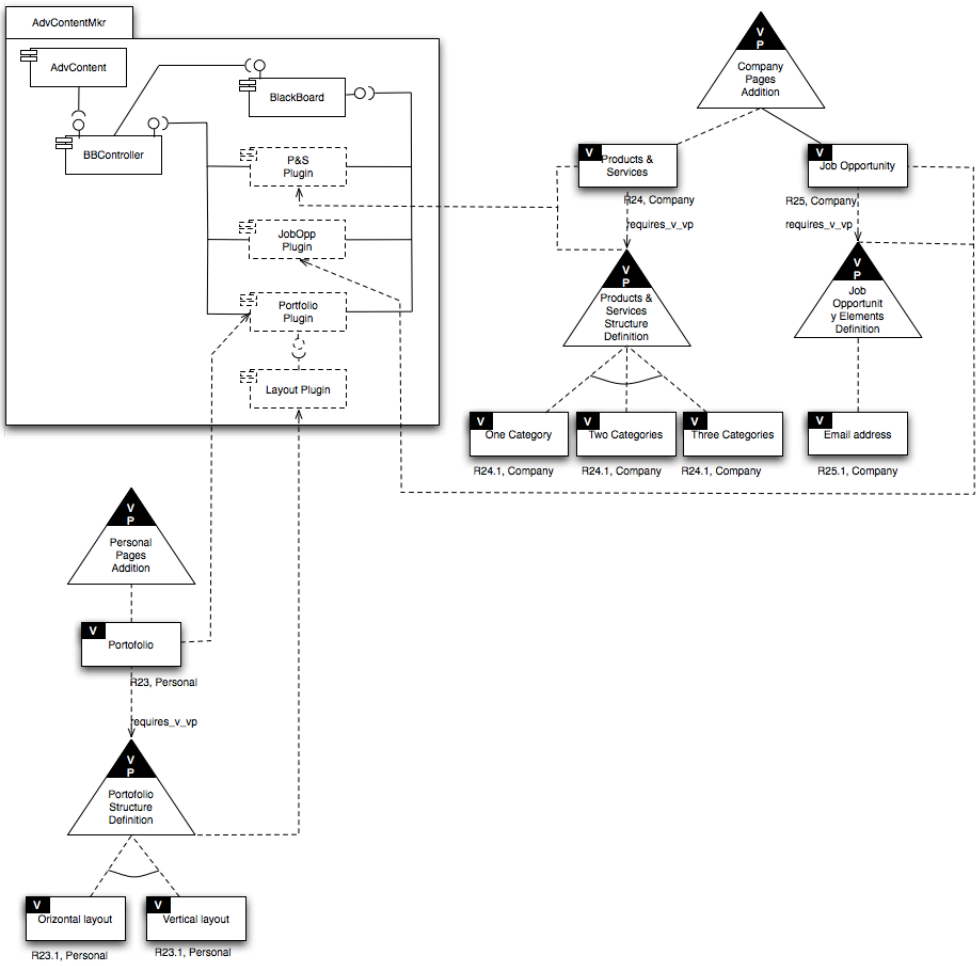


Figure 4.19: Variability in AdvContentMkr component

In addition to the static models of component and variability diagrams so far discussed, the collaboration diagram among the different components is introduced (n.b. despite from UML standards collaboration diagrams refer to objects of classes, we preferred to refer to components to facilitate the readers with a holistic picture of our system's functionalities).

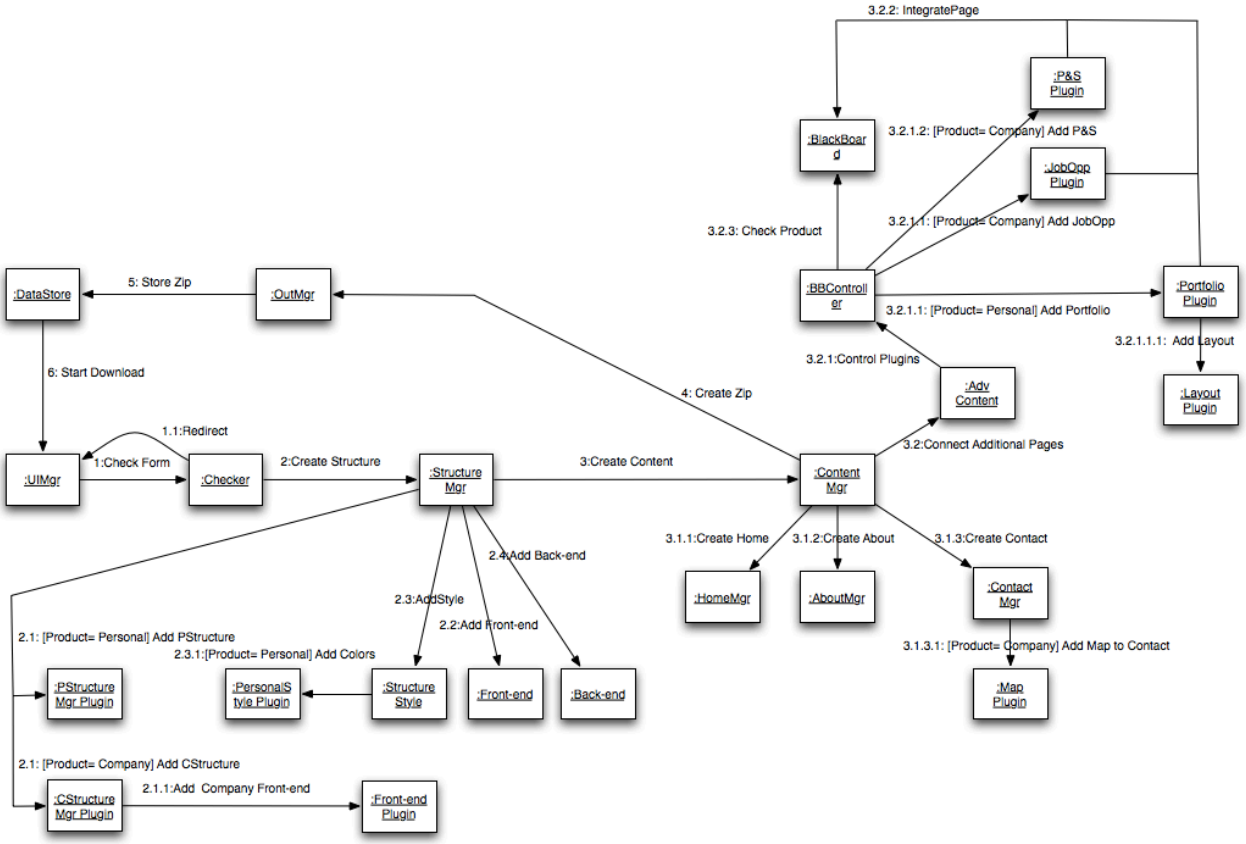


Figure 4.20: Collaboration Diagram

The collaboration diagram shows how from *UIMgr* it is possible to start the construction of the template by sub-activities of *StructureMgr* related to additional plugins and family products components operations. Then, the *ContentMgr* is responsible to interact and active the black-board pattern for the additional contents of previously specified pages. At the end the *OutMgr* properly creates the zip file to be stored and sent to the customer for starting the download.

Mapping the components to the related modules and classes should be conducted to provide module diagram and class diagram. *Table 4.12* presents a sample of this mapping. Defining special naming rules, will improve the readability of design. As an example of naming rules, name of modules can be started with “M”, functional classes with “C” and interfaces with “I”.

Component	Module	Class
UIMgr	MUIMgr	IUIMgr IRedirect CUIMgr CUIPresentMgr CRedirect
	MUIStyleMgr	CUIStyleMgr
Checker	MChecker	IChecker CChecker
StructureMgr	MBoxMkr	IBoxMkr IPStructure ICStructure CMenu CBoxMkr CPStructMkr CCStructMkr

Table 4.21: Component Mapping to the Modules and Classes (an example)

Based on the above mapping, it is possible to provide models for modules and classes. As the big diagram from all classes of *Web-line* may not be readable, the cropped image containing the selected modules classes and interfaces are illustrated in *Figure 4.22* .

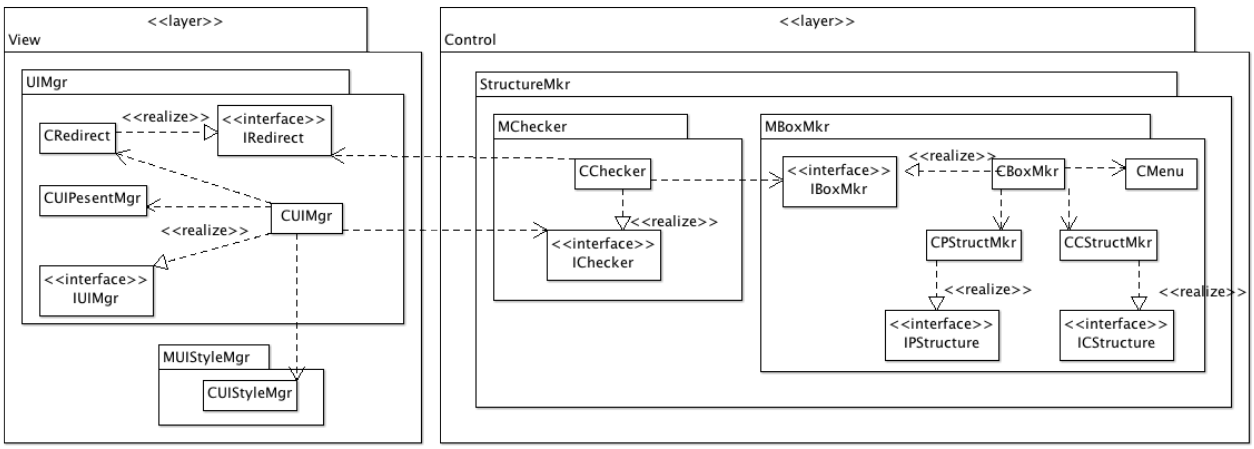


Figure 4.22: Class Diagram (an example)

Validating the architecture

Validation of the architecture is conducted by validating the software assets of the reference architecture. In order to validate it the following questions should be answered [3]:

- Do interfaces carry the right functionality to the right level of abstraction?
- Are components and interfaces produced according to used specific solutions such as styles or design patterns?
- Does each component carry all its interfaces, and no more? Have the provided interfaces (

 and required interfaces () been used in the correct way?

- Do components call only the required interfaces, and all of them?

4.4 Domain Realization

Domain realization is the sub-process related with the design and implementation of reusable software assets, based on the reference architecture. This is the where the actual implementation of the platform takes place.

The *Figure 4.23* shows the relations with other sub-processes. Domain realization gets the reference architecture and reusable software artifacts from Domain Design (*Step 1*) and then designs and implements the corresponding reusable artifacts. When issues arises during the domain realization, problem reports are sent back to the Domain Design (*Step 2*).

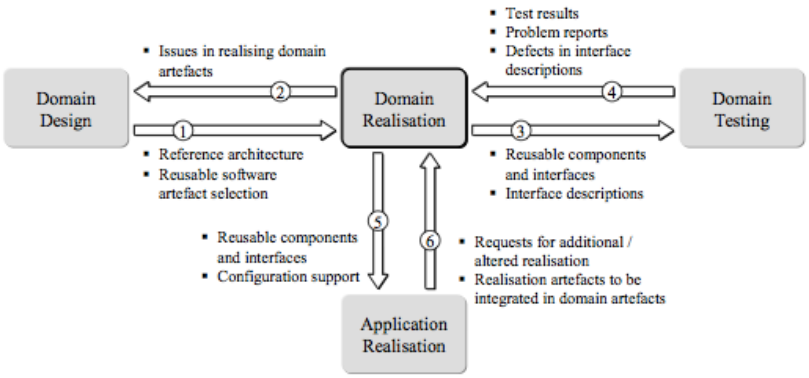


Figure 4.23 : Information flows between domain realization and other sub-processes

Domain Realization also sends the implemented reusable components and interfaces to the Domain Testing (*Step 3*) to be checked. While testing is conducted in Domain Testing, test results and reported problems are returned to the Domain Realization to be fixed (*Step 4*). In *Step 5,* the implemented reusable components and interfaces are sent to the Application Realization which builds one application by using the domain reusable components and interfaces. Also configurations are submitted to the Application Realization in order to assemble new application. Any request additional changes in design are sent back to the Domain Realization in *Step 6*. Traditional realization contains the following activities:

- Detailed design
- Interface design
- Interface implementation
- Components implementation
- Compilation

Example : Web-line Realization

In the following section we will give a brief overview on how we built our *Web-line*, trying to give a good high-level overview to the reader. In some specific parts that we consider important we will enter into technical details. The source code we refer to is released under GPL license and available at

[15].

//Development Environment

Web-line product line is a web-application based on the Google App Engine (GAE) work-frame [6]. They support java and python, and we chose to go with java and use their free offer that consist of:

- Good set of APIs and services
- Web Hosting
- Database
- Cloud resources (both for computation and deployment)
- An SDK for *Eclipse*

We use *Google Code* as repository as they offer:

- SVN version control system
- Issue tracking
- Wiki

The *Web-line* platform is written using different well-known languages:

- Java (servlet and jsp)
- Javascript (with ajax and jquery)
- Cascading style sheets (CSS)
- HTML 4

Our product line and the products that we generate are accessible on from any modern browser at [16]. The platform we built is adapted to the GAE pattern architecture (MVC-like) shown in the picture below.

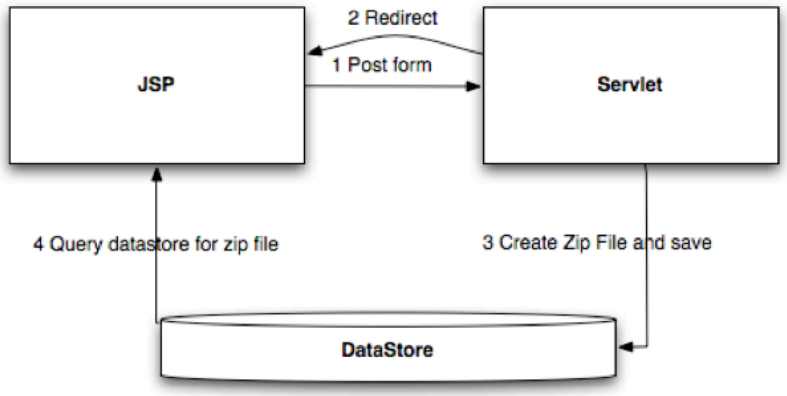


Figure 4.24 : GAE compliant architectural style

//Product-line flow

Our platform consists of a front end (*jsp*) and a back end (*servlet*) that communicate through an interface layer. The next *Figure* gives to the reader a bird-eye view of the dynamic behavior of our product-line, that can be followed in circle starting from the customer.

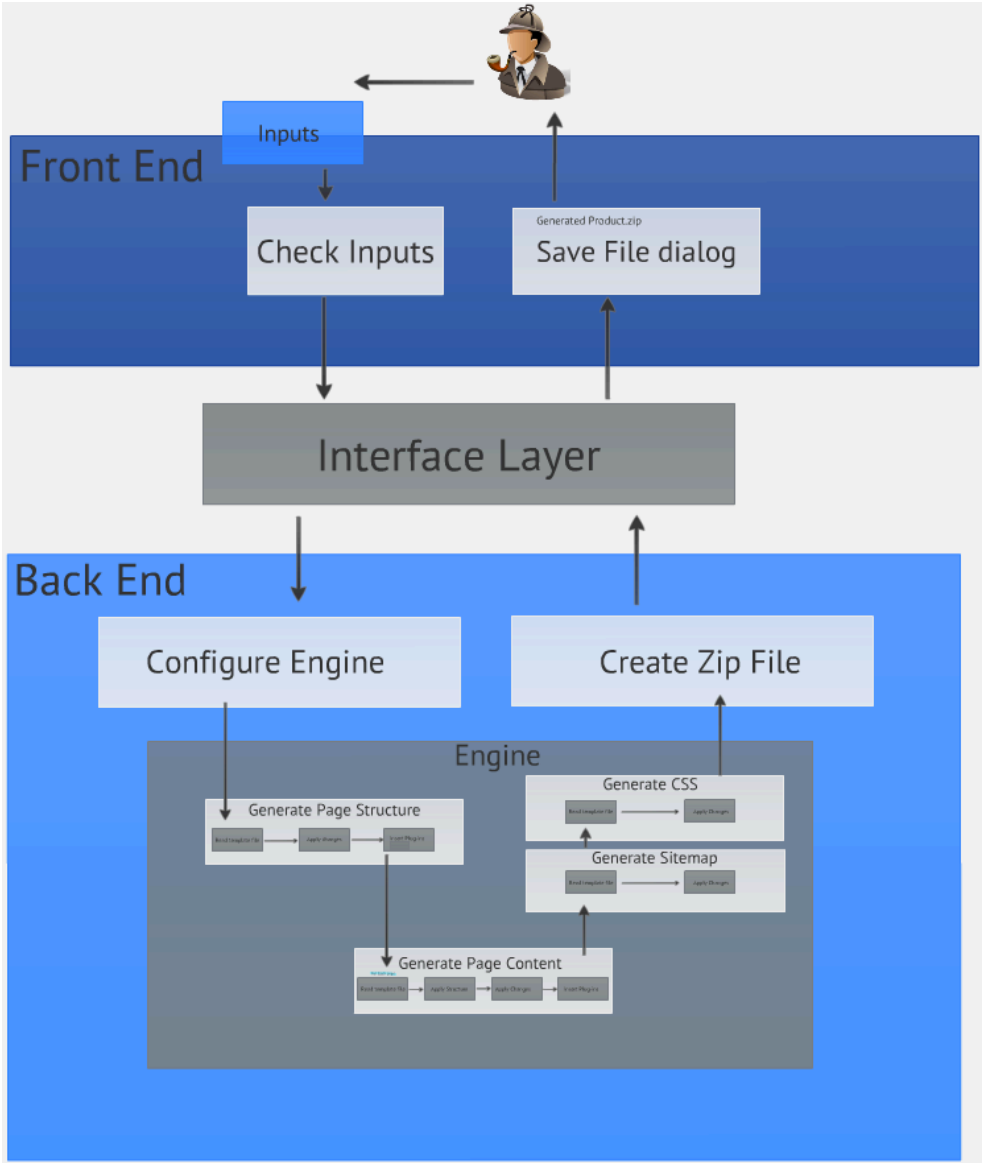


Figure 4.24: Web-line Dynamic functioning overview

As shown in the picture above, the customer selects the options in the front end and send them through post (*http request*) to the back-end (a *servlet*) . The following screenshot partially shows the configuration view where the customer can input options in order to customize the product.

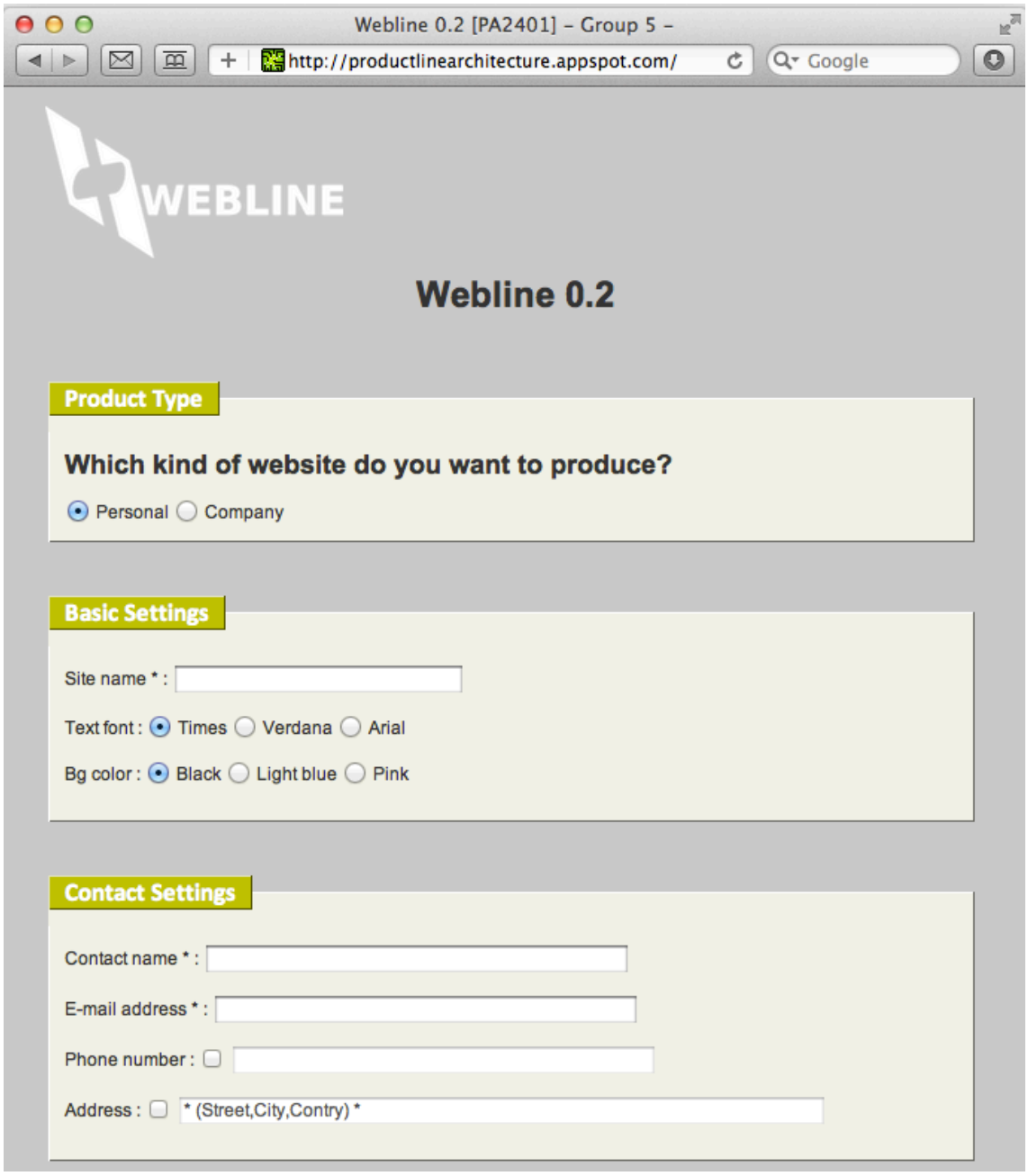


Figure 4.25: View layer example

There are many options the user can access in the form of html inputs (*textfields*, *checkboxes*, r*adio buttons* etc.) and then finally a '*Create Product'* button that is shown in the next figure.

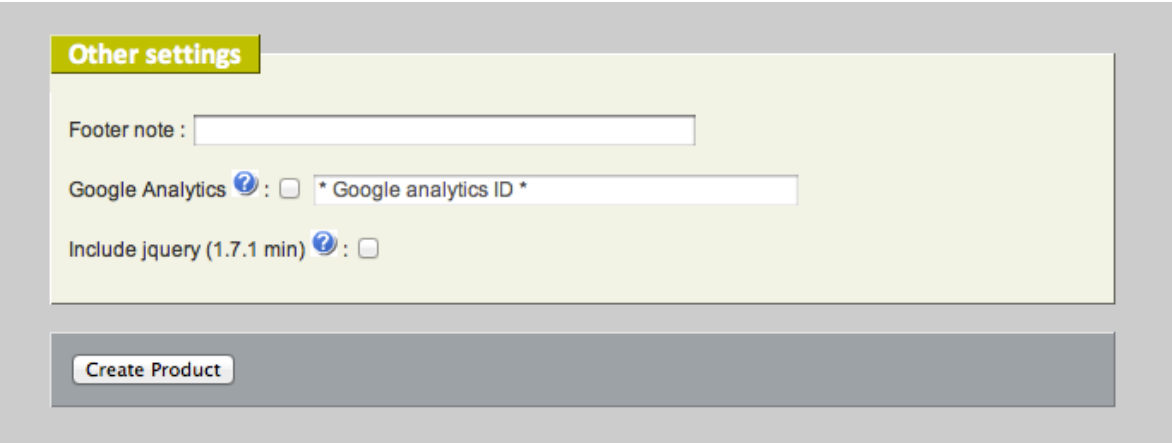


Figure 4.26: Create Product button, in the view layer.

The input data is therefore checked and passed to different components until it finally reaches the *Engine* (figure below), that is where the product is assembled.

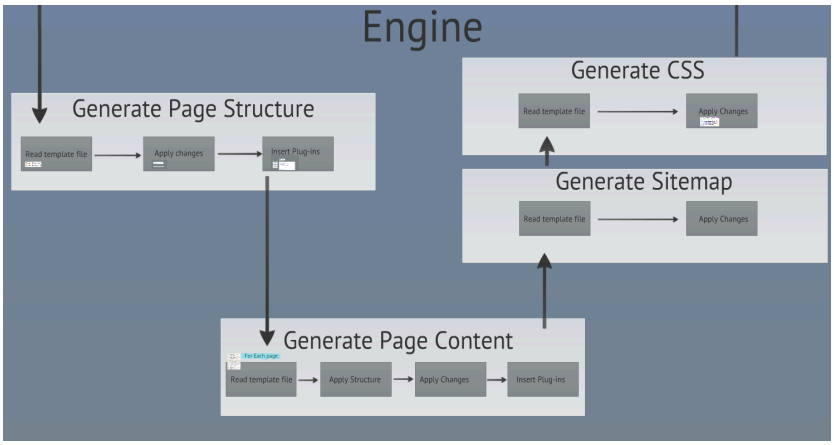


Figure 4.27 : Engine Overview

Here in the engine is where we apply the variability. Since what we deliver as final product is basically a collection of static source files, to configure the final product we have to edit the source code of the application in a dynamic way before delivering it. We have two different mechanisms to perform this operation : through the use of plug-ins or through parameters injection.

We created a set of templates files, that we load dynamically into the back end . Then we scan through the content of each file looking for a special tag to parse according to user's preferences. The following picture shows the list of our static templates files in the *war*.

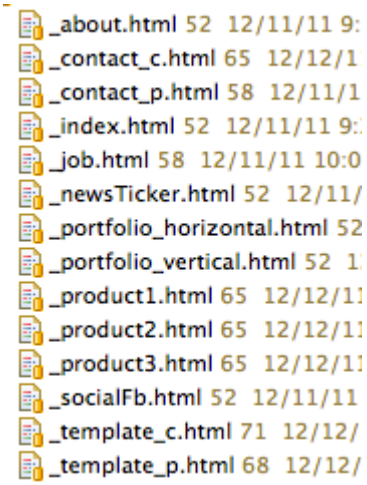


Figure 4.28 : Template files

In the next slice of code, you can see an example of what it is inside a template file : is basically a static html file with some special *tags* that are the entry point for our variability. The following example represents the *contact box* on the sidebar of the company website.

```
<!-- Contact Box -->
<div class="sidebar-widget">
    <h4>Contact Us</h4>
    <p>
    <strong>Name: </strong> <!--#WPL_ID='contactName'--><br />

<strong>Email: </strong><a href="mailto:<!--#WPL_ID='contactEmail'-->"><!--
#WPL_ID='contactEmailRef'--></a><br/>
    <!--#WPL_ID='checkPhone'-->
    <!--STOP-->

    <!--#WPL_ID='checkAddress'-->
    <!--STOP-->
   </p>
</div>
```

The special tags are in the form of an HTML comment with the special unique id that allows the parser to know the position of the tag inside the template.

```
<--!#WPL_ID='<id-goes-here'>-->.
```

In the platform we seek for these special tags and we replace it with the content chosen by the customer. We built a parser to realize the replacement, that we can access using the method *parse*.

```
parse(String originalString,String newText,String position-id, String mode) : String
```

The next code snippet shows the example of java code related do the parsing of the template presented above.

```
//Contact name
templateString = parser.parse(templateString, common_params.getCname(),
                 "contactName",Parser.REPLACE);
//Contact email
templateString = parser.parse(templateString, common_params.getCmail(),
```

```
                    "contactEmail",Parser.REPLACE);
//Contact email href
templateString = parser.parse(templateString, common_params.getCmail(),
                    "contactEmailRef",Parser.REPLACE);

//Contact Phone
if(common_params.isCphoneCheck())  //Did the customer selected this option?
  {
   templateString = parser.parse(templateString,
   "<strong>Phone number:</strong><!--#WPL_ID='contactPhone'--><br />" ,
   "checkPhone",Parser.REPLACE_TILL_STOP);

   templateString = parser.parse(templateString, common_params.getCphoneValue(),
                           "contactPhone",Parser.REPLACE);
  }
```

For include a plug-in the mechanism is similar : we seek for special tags inside the template file, like in the snippet below.

```
if(company_params.isMapCheckBox()) //Did the customer selected this option?
  {
   //Create input parameters for the plugin
   String[] params = {company_params.getCaddressValue(),
                      company_params.getCaddressValue()};

   //Create the object to manage plugins
   IAdvContent iacMap= new AdvContent();

   //Create the map plugin
   iacMap.addPlugin("MapPlugin", params);

   //Update the template with the content of the plug-in
   templateString = parser.parse(templateString,iacMap.createAdvContent(),
                           "gMap",Parser.REPLACE_TILL_STOP);
 }
```

In the next section we will give detail explanation on how the plug-ins mechanism work.

//Plug-ins integration

Generally, there are three ways to work with plug-ins into eclipse:

1. A plug-in consists of one .jar file placed in a specified folder and it comes with a configuration file.
2. The “classes” folder and all jars are located in a specified folder.
3. Plug-in files are located in a single jar-file which is placed in a specific folder.

In our project we implemented the second choice where all the jars are located in a specific folder (named *plug-ins*) that is included in the .war file deployed on the web server.

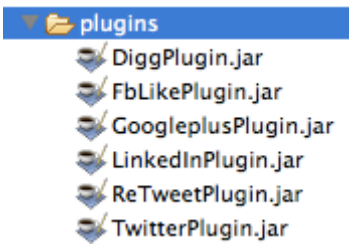


Fig. 4.29 : Plug-ins files

The different plug-ins have supported the creation of part of the html code of the final product by providing specific text-codes. In order to increase decoupling among the product line and the introduction of the plug-ins, each jar has been created by providing the same signature method that is called "*getPluginCode()*". In the follow figure we can see the example of the *Digg* plug-in source code.

```
public class DiggPlugin {
  private static final String[] pluginCode = {"<script type='text/javascript'> \n" +
     "(function() { " +
     "vars = document.createElement('SCRIPT'),s1 =
            document.getElementsByTagName('SCRIPT')[0];\n" +
     "s.type = 'text/javascript';\n" +
     "s.async = true;\n" +
     "s.src = 'http://widgets.digg.com/buttons.js';\n" +
     "s1.parentNode.insertBefore(s, s1);\n" +
     "})();\n" +
     "</script>\n" +
     "<li><a class='DiggThisButton DiggMedium'></a></li>\n" +
     " <li></li> "};

    public static String[] getPluginCode(){
        return pluginCode;
    }
}
```

Applying this rule, we had the possibility of calling jars iteratively using *java-reflection.* Nevertheless, it was important knowing position and parameters of each of them as previously explained. An overview of the plug-in structure is presented in the next figure.

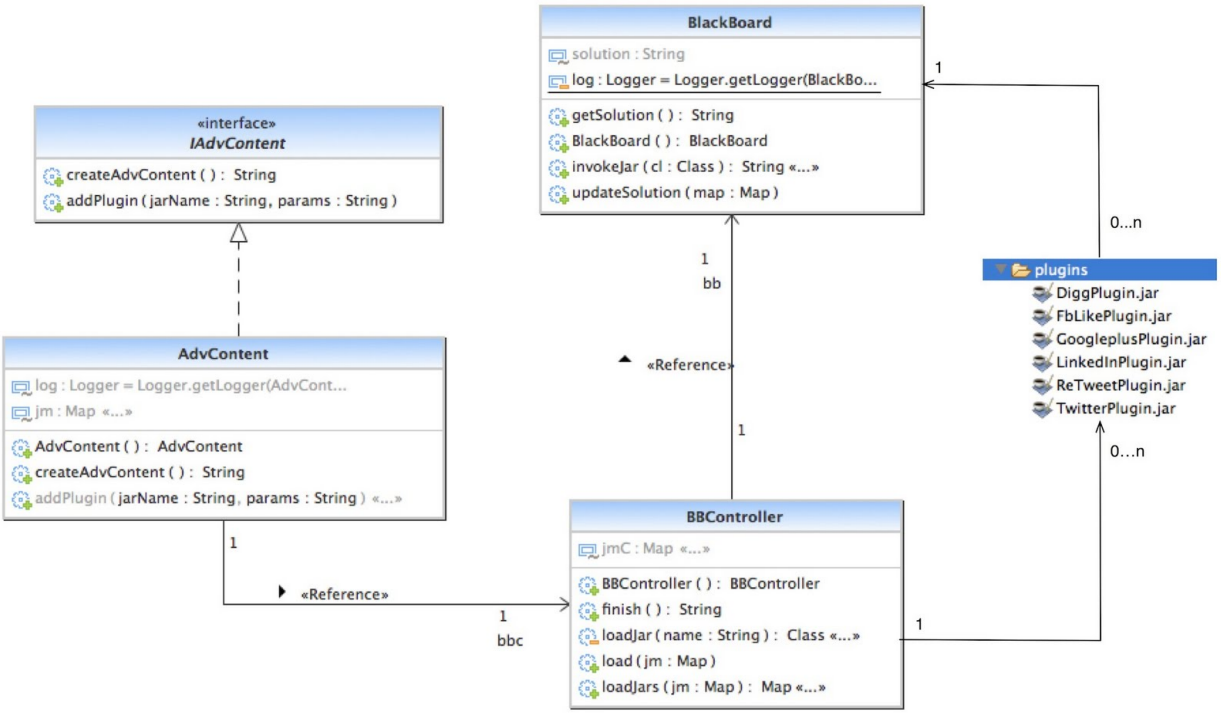


Figure 4.30: An example of Black-board pattern for plug-in integration

As we can notice, the presented class-diagram follows the Black-board design pattern [7]. The advantage of such pattern is that all the plug-ins can be easily updated, and adding new ones doesn't affect the overall architecture . The interface of this pattern is *IAdvContent,* that offers two services: add plug-ins providing their name and parameters; create the final content of the html code. Then,

BBController will be in charge of loading the different plug-ins in run-time, which will create the final content in the *BlackBoard* class invoking the jars and updating the final solution (for more details see the section of domain design engineering).

4.5 Domain Testing

Domain Testing aims to validate the output of the other Domain Engineering sub-processes through system testing. In *Web-line*, system testing is developed starting from the domain requirements artifacts and variability model. To facilitate the design of our test cases, we adopted the strategy of dividing system testing in two parts: the part of the system that realize common requirements that's not affected by the variability of the software product line and the part related to variability only.

Moving to domain design, we developed an integration test using the reference architecture for validating the interactions among components. It was straightforward to realize such interactions since most of them are directly defined from the use-case in the domain requirements engineering. Within domain realization, testing consisted of creating input samples for catching *corner cases* and *edge cases* [9] through unit testing: at the class level of our project, all the interfaces and reusable components were tested by specific parameters within the set of valid and non-valid inputs (in presence of numeric variables, boundary cases were considered as well).

For creating tests we followed the *test dependency model* (see following figure): for example, in the integration test, to define a relationship between tests and components, we referred a test to a requirement (i.e. any kind of specification) to validate a single component.

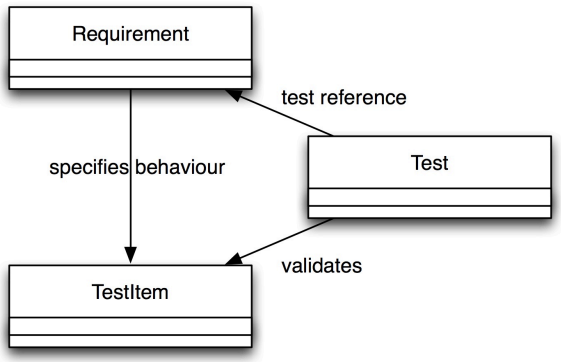


Figure 4.31: Test dependency model

To validate the domain engineering realization, the *sample application strategy* [3] has been followed. It consists in creating one sample application with all the commonalities and one particular configuration of variability to verify the behavior of commonalities of the whole product-line and the binding mechanisms of the different features.

5. Application Engineering in Practice

Application engineering aims to provide applications by reusing as many domain artifacts as possible. Our case study presents two families of products, each of them with several variants. To simplify the

explanation of this section, only one example will be reported. All the sub-processes will follow the specification of the application engineering shown in the following figure.

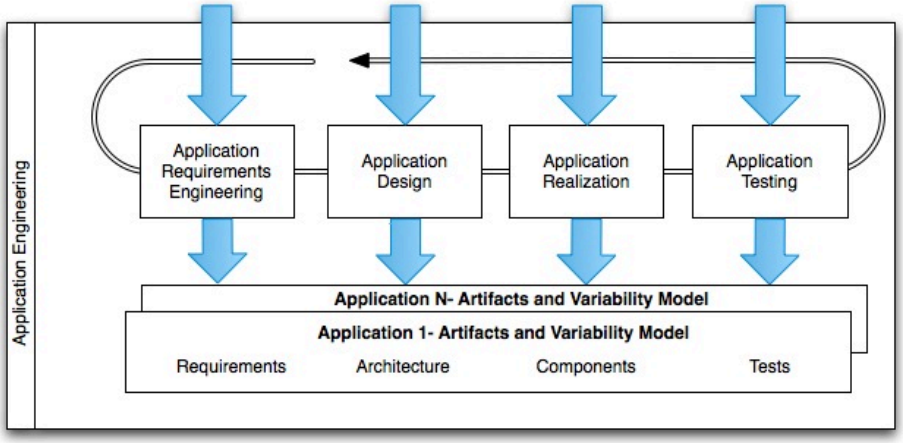
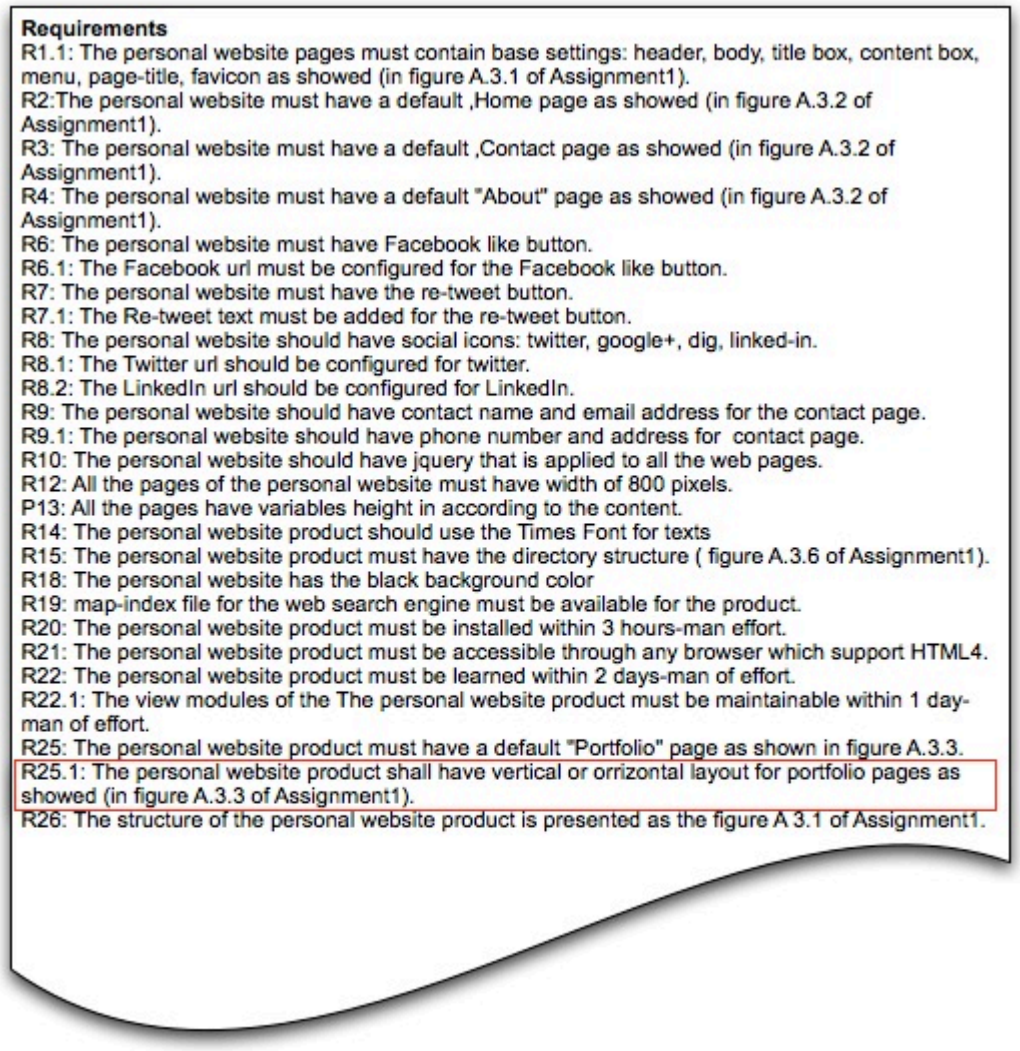


Figure 5.1: Application engineering sub-processes

5.1 Application Requirements Engineering

The first step of application requirements engineering is related to the product management interaction. During this phase all the features related to a specific application are taken into account and new features are altered in according to stakeholders' needs as well. In the course of the development of this case study the following feature has been modified: "*The personal portfolio web-page has been altered with the introduction of the vertical layout.*"

In order to introduce this new feature, the variability models, common and variable requirements artifacts needs to be defined. Requirements have been provided for defining a specific product. Within a red rectangle the new requirements related to the vertical layout has been presented. It is called requirement *delta* [3] since it is born by the definition of this application.



Requirements
R1.1: The personal website pages must contain base settings: header, body, title box, content box, menu, page-title, favicon as showed (in figure A.3.1 of Assignment1).
R2:The personal website must have a default ,Home page as showed (in figure A.3.2 of Assignment1).
R3: The personal website must have a default ,Contact page as showed (in figure A.3.2 of Assignment1).
R4: The personal website must have a default "About" page as showed (in figure A.3.2 of Assignment1).
R6: The personal website must have Facebook like button.
R6.1: The Facebook url must be configured for the Facebook like button.
R7: The personal website must have the re-tweet button.
R7.1: The Re-tweet text must be added for the re-tweet button.
R8: The personal website should have social icons: twitter, google+, dig, linked-in.
R8.1: The Twitter url should be configured for twitter.
R8.2: The LinkedIn url should be configured for LinkedIn.
R9: The personal website should have contact name and email address for the contact page.
R9.1: The personal website should have phone number and address for contact page.
R10: The personal website should have jquery that is applied to all the web pages.
R12: All the pages of the personal website must have width of 800 pixels.
P13: All the pages have variables height in according to the content.
R14: The personal website product should use the Times Font for texts
R15: The personal website product must have the directory structure (figure A.3.6 of Assignment1).
R18: The personal website has the black background color
R19: map-index file for the web search engine must be available for the product.
R20: The personal website product must be installed within 3 hours-man effort.
R21: The personal website product must be accessible through any browser which support HTML4.
R22: The personal website product must be learned within 2 days-man of effort.
R22.1: The view modules of the The personal website product must be maintainable within 1 day-man of effort.
R25: The personal website product must have a default "Portfolio" page as shown in figure A.3.3.
R25.1: The personal website product shall have vertical or orrizontal layout for portfolio pages as showed (in figure A.3.3 of Assignment1).
R26: The structure of the personal website product is presented as the figure A 3.1 of Assignment1.

Figure 5.2: An example personal website product- requirements

When new variants or constraint dependencies are modified, analysis of *delta* requirements are defined. In our case an invariant part must be turned into a variable part. Precisely the external variability of the portfolio page structure needed to be introduced and extended by the vertical layout variant (see figure 4.9, where *portfolio page* variant was only related to a fixed vertical structure). Therefore, from this analysis we needed to introduce a new variant, which also required the definition of a new variation point for *portfolio structure* as showed in the next figure.

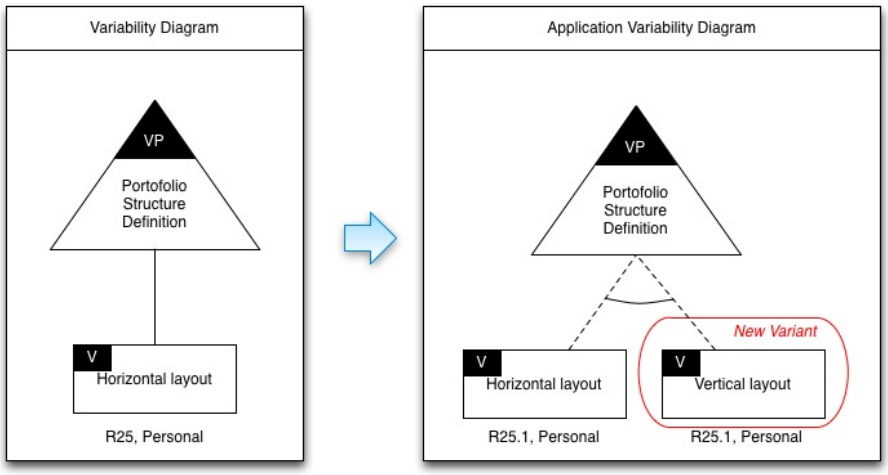


Figure 5.3: Addition of the vertical layout due to a requirement delta

As defined by the requirement *R25.1*, the variability model had to provide a unique choice among the two layouts: horizontal and vertical. The transition from the left variability diagram to the right one shows how it is possible to transform a fixed variant definition in two optional ones, that in our case are mutually exclusive. The integration with the rest of the variability model is showed in figure 5.5.

This delta analysis will be integrated within the product management with the new feature and in requirements engineering with the new delta requirement. The effort to introduce new variants is different. In this case the effort is moderate since only a new variant need to be adapted in the variability model. In more general cases the relation between deltas and architectural adaptation effort categories is described by the following table.

Category / Delta	Category A **No** adaptation effort	Category B **Moderate** adaptation effort	Category C **High** adaptation effort	Category D **Too high** adaptation effort
1. New variant	X	X		
2. Adaptation of variability dependencies	X	X	X	
3. Adaptation of constraint dependencies		X	X	X
4. New variation point		X	X	X

Figure 5.4: Architectural adaptation effort categories [3]

Later on, the requirement is developed within the general variability model defined in the Domain Engineering.

In this example of personal web-site, the products provides specific variants that are: personal index, black background and portfolio page with vertical layout. As showed in the next figure, the application binds different variants (contained within red circles) defined by the previously variability model.

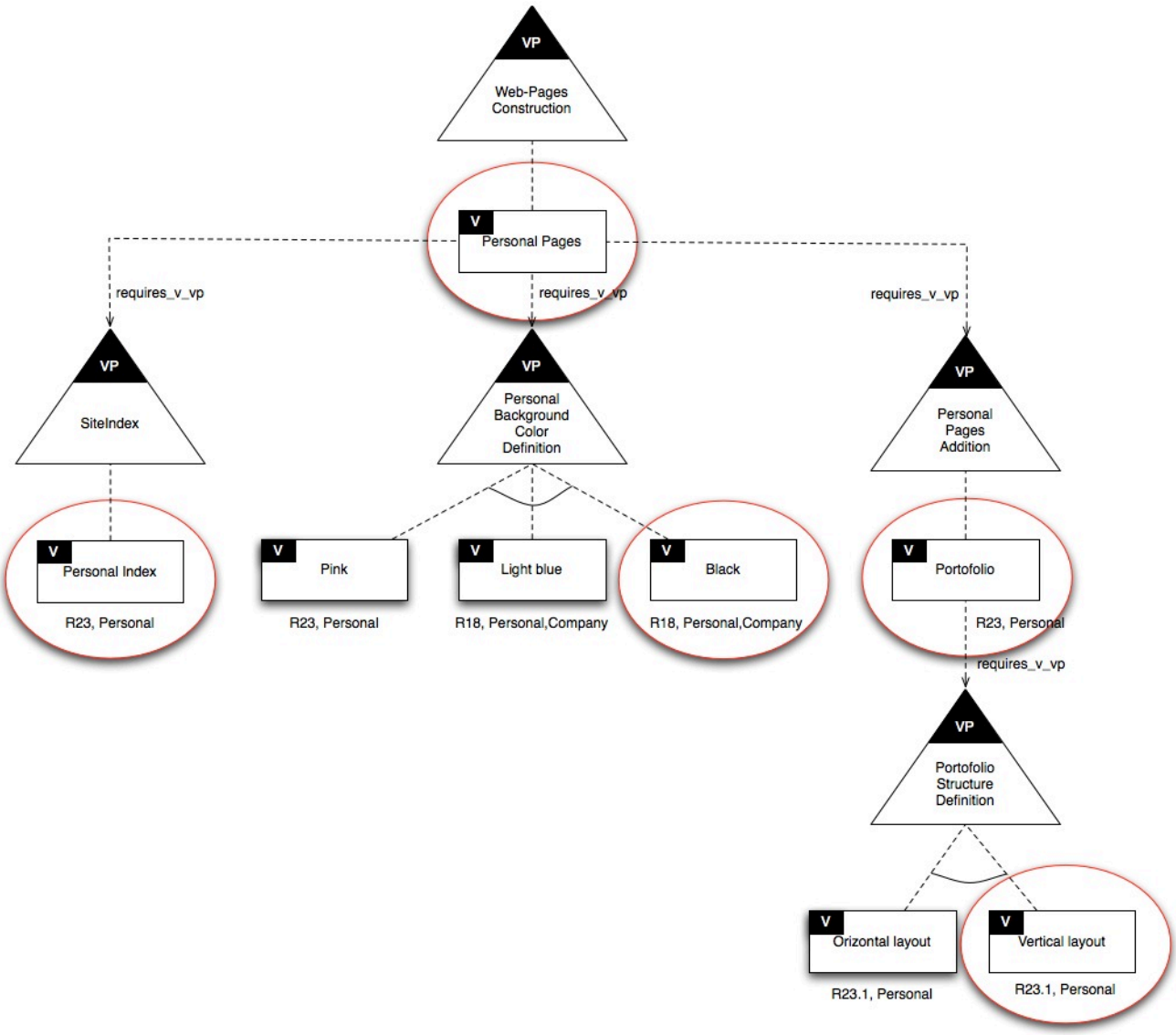


Figure 5.5: An example personal website product- requirements

During the process of application requirements an additional requirement has been needed to increase the chance of user's satisfactions. Finally, requirements specifications have been passed to the Application Design, which is going to be detailed in the next section.

5.2 Application Design

Application architecture is a specialization of the reference architecture developed in domain engineering. In this sub-section we are going to bind the variability previously introduced, showing how it is related to the component diagram. In general the reference architecture determines the easiness of binding variants since the additional work is related to developing application-specific

variants. In our study the amount of application-specific variants have been developed during the application sub-processes life-cycle and, afterwards, introduced in the product line to enhance re-usability (see issue in *data collection* section).

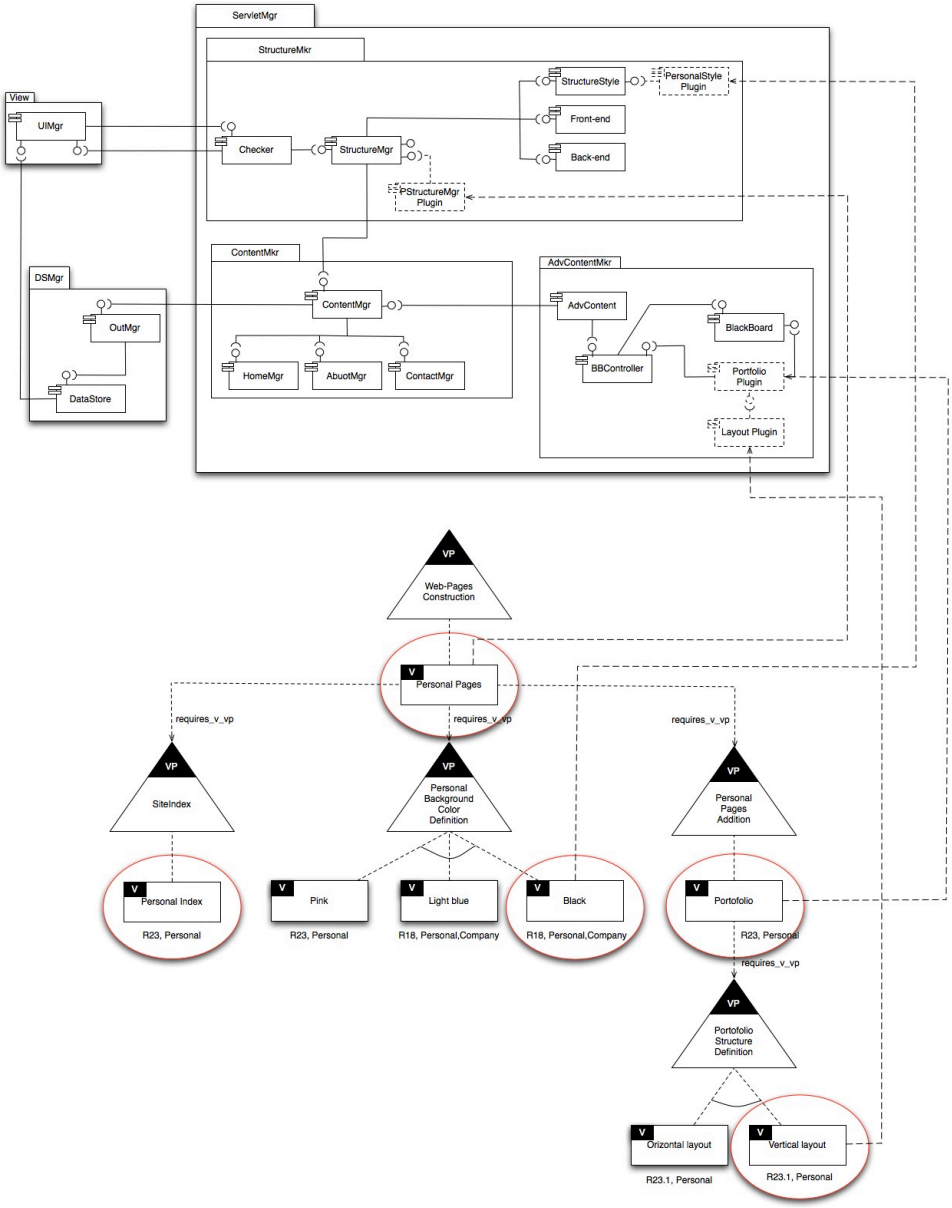


Figure 5.6: Variability diagram to Application Architecture

From this configuration we are able to build our final product, that is presented with more details in the next section.

5.3 Application Realization

As we already mentioned before, our applications are composed by a set of interconnected static files: html pages, cascade style sheets, javascript, and images. After the different file are customized, assembled and linked together in the product line, the application is zipped and returned to the customer. In the next figure we provide an example of the file-structure that forms a personal website (P1) .

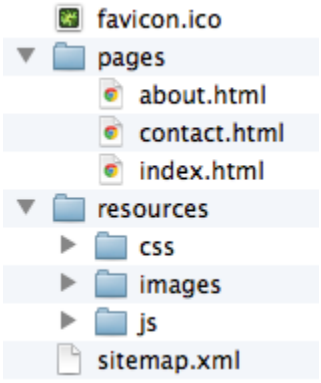


Figure 5.7: An example personal website product - directory structure

The *main* page is contained inside the file `pages/index.html` which can be opened and surfed with any modern browser. Relations among html pages and the other files (*resources*, *favicon*, *sitemap*) is embedded in the html code, as shown in the example below :

```
<!-- ## Example : Include a javascript source file ##  -->
<script language='javascript' src='../resources/js/jquery.js'></script>

<!-- ## Example : Include a cascade style sheet ##  -->
<link href="../resources/css/black.css" rel="stylesheet" type="text/css" />

<!-- ## Example : Include an image ##  -->
<img src="../resources/images/logo-1.png" alt="WEBLINE" /></a>

<!-- ## Example : Include the favicon##  -->
<link rel="shortcut icon" href="../favicon.ico" />

<!-- ## Example : Link to another HTML page ##  -->
<li><a href="about.html"><span>ABOUT</span></a></li>
```

The architecture of our products is therefore determined by aggregation of files and their relations and communication path. In the image below we provide an example of architecture for a *personal website* (P1) .

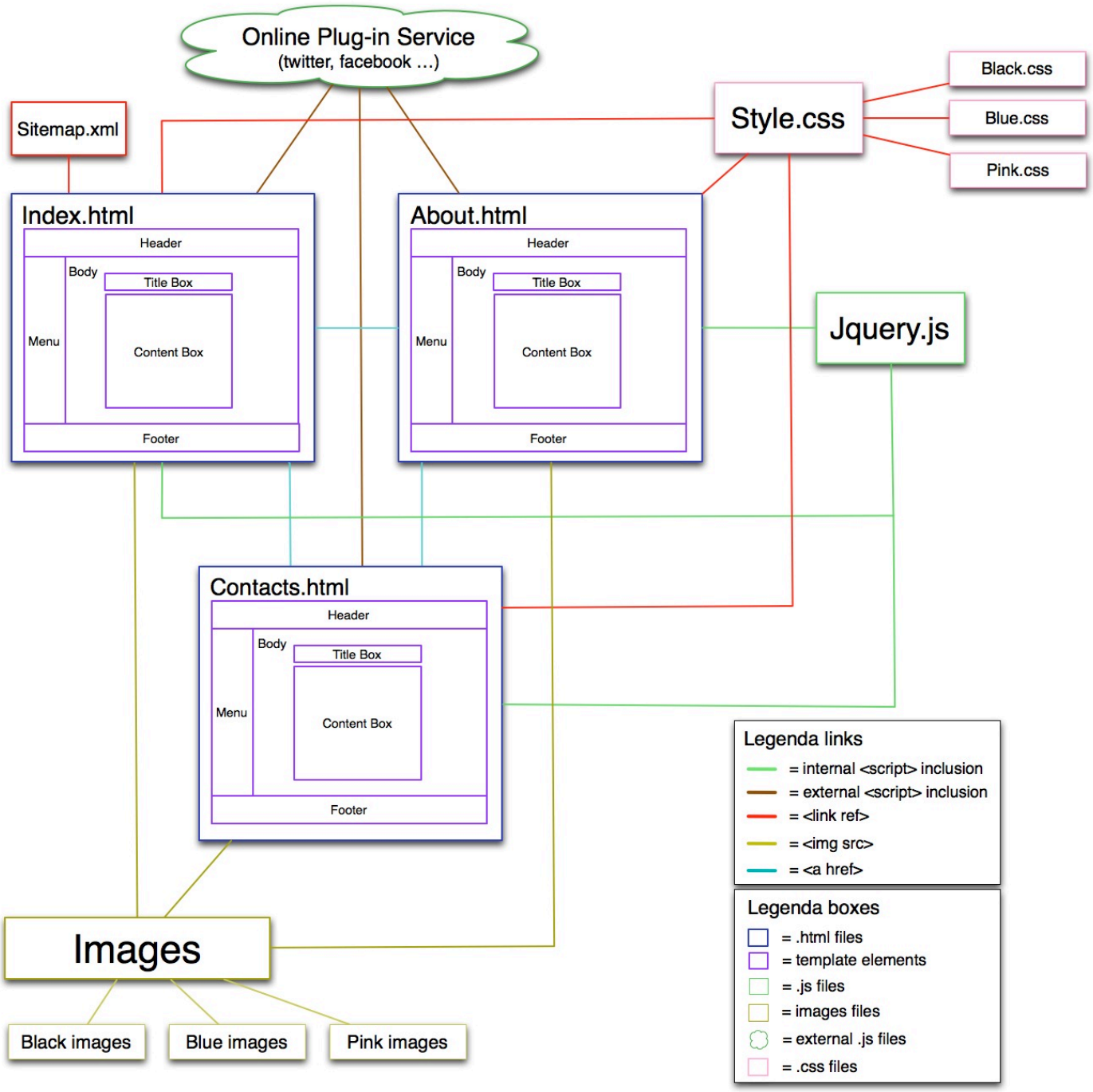


Figure 5.8: Product Architecture overview (personal website)

For the sake of brevity we don't want to go into detail for each file that compose our web site as is not really relevant for understanding product-lines. The resources and the source code itself is well structured and commented and it's up to the reader curiosity to read through it . It's enough for the reader to know that our product is w3c compliant. The customer (*f.i.* a webmaster) can open html files with a web-design tool and edit its content, that we design to be easy accessible, and structured enough to be *manually* customized without too much effort. In the next figure we show some screenshots of how our final product looks like in a browser.

Figure 5.8 : An example personal website product, different pages.

The application has been developed binding additional personal-website plug-ins to the already predisposed ones in the product-line platform (see *Fig.4.30)*.

Considering that for personal-website (P1) we don't have extra plug-ins, as example we demonstrate how the black-board pattern would behave in case of a company-website product in the following image.

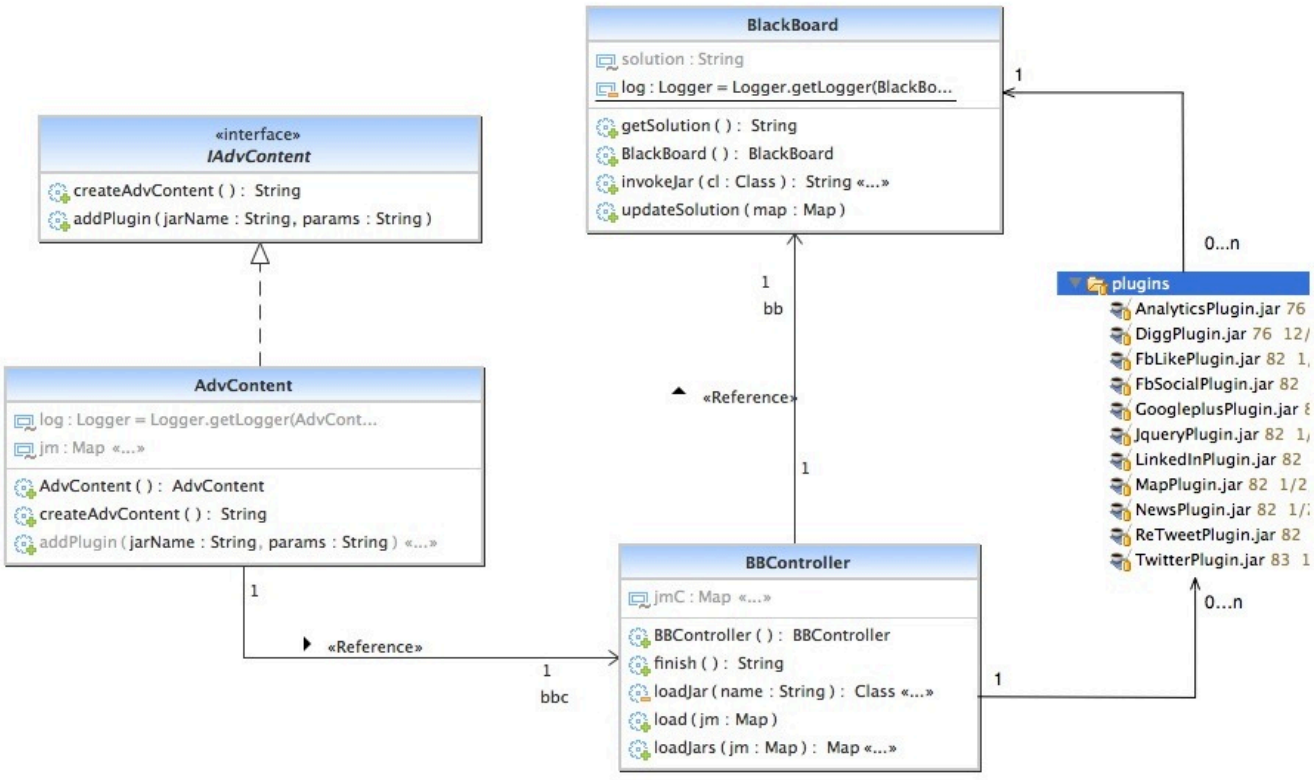


Figure 5.8: Black-board pattern for the personal web-site plug-in integration

As showed by the figure above, a company web-site product integrates plug-ins enriching the deployed library (*plugins* directory) with a minimum additional effort.

5.4 Application Testing

From requirements specifications, architecture and interface descriptions testing are carried out. All the results are useful to detect defects in the different sub-processes of application engineering. In order to perform the tests, we had to bind the variation points to the platform for reusing test artifacts of the domain test design, as suggested by [3].

In order to avoid developing tests from scratch, we reused the tests designed for the Domain Engineering, adding to them new bindings of the application variability model. Any defects reported in the results were traced back to Domain Testing and corrected.

Two main tests were performed: *variant absence test* and *application dependency test*. *Variant absence test* verified that no more than the variants selected from the customer were added to the final application. *Application dependency test* verified that the application is in conformance with constraints and dependencies specified in the variability model.

The main purpose of the application testing is to verify that the application produced by the product-line is coherent with the initial settings that the customer entered. Since we have many variations, in order to facilitate the testing process we encapsulated all the inputs from the customer into objects before processing them.

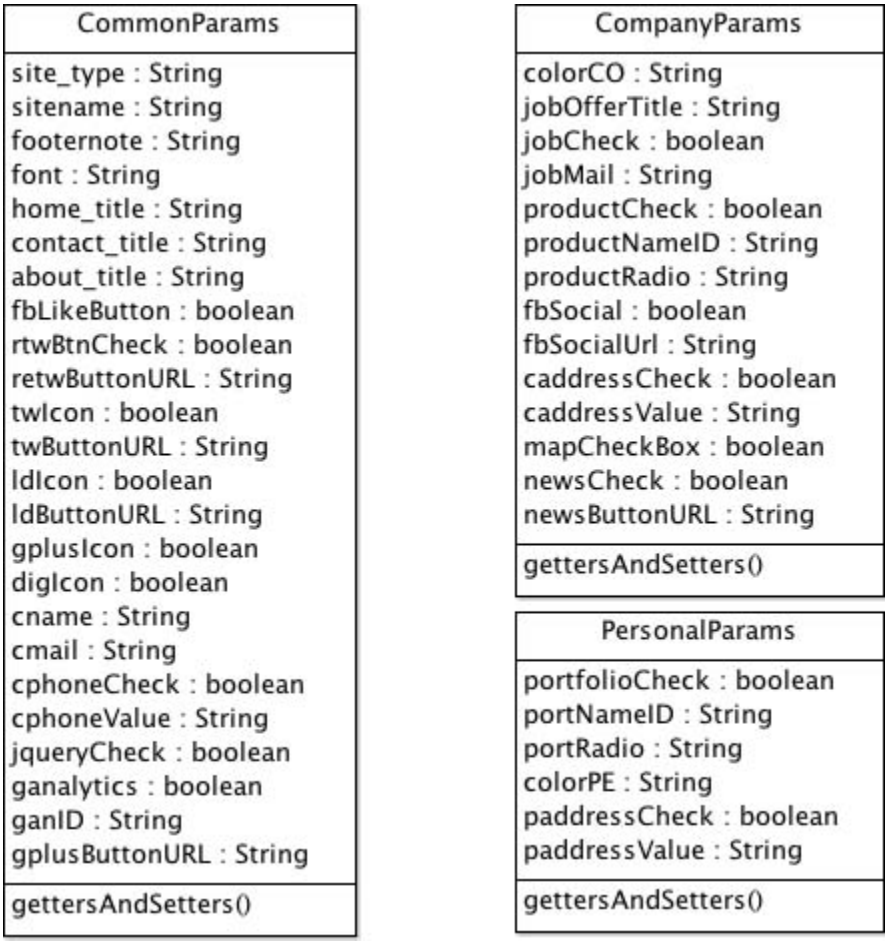


Fig. 5.9 : Class diagram of the object which encapsulate customer inputs

Then we created a container object to handle the common and non-common parameters together, as shown in the next figure.

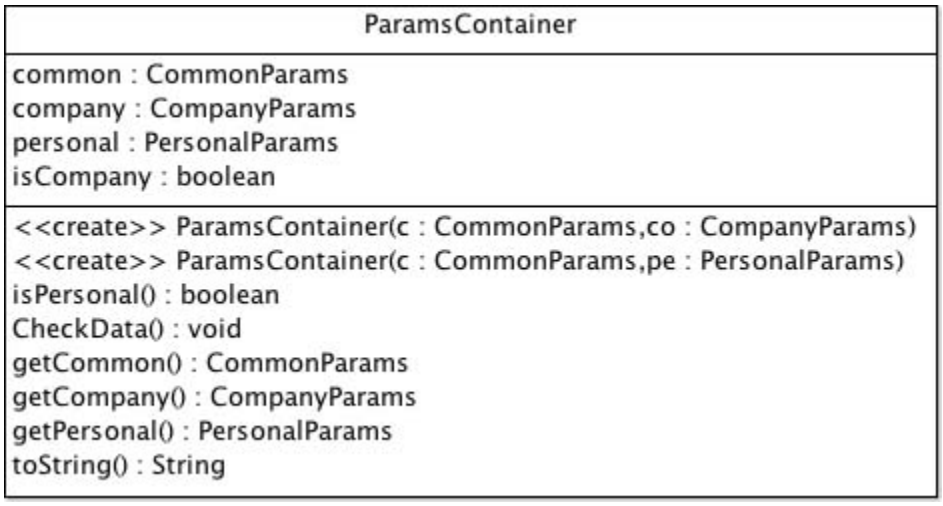


Figure 5.10 : Objects embedding all user parameters

And to finally test the product, after we build the product, we scan the generated code seeking

for inconsistencies between what the customer selected and what is actually embedded into the application.

6. Product-line Project Management

In traditional software engineering projects, project management focuses on conducting the team for producing a specific product. This conventional aspect is turned upside-down since SPLE provides many product definitions within one project delivery. In this section we show how a proper SPLE Management process is crucial in order to succeed.

6.1 Planning

The project planning is pertained to activities of technical and organizational management practices, that we had to conduct to accomplish the defined project goals and objectives.
First of all, all the artifacts of our SPLE were approached as *configuration items* where the activities aims to identify, produce, store and release them for usage. The tools that supported those activities are: *google docs* to document the artifacts of the sub-processes; and *subversive* for tracking issues, changes and integrations of different development versions.

Moreover, all the milestones have been traced on a *Gantt chart* to highlight the different achievements.

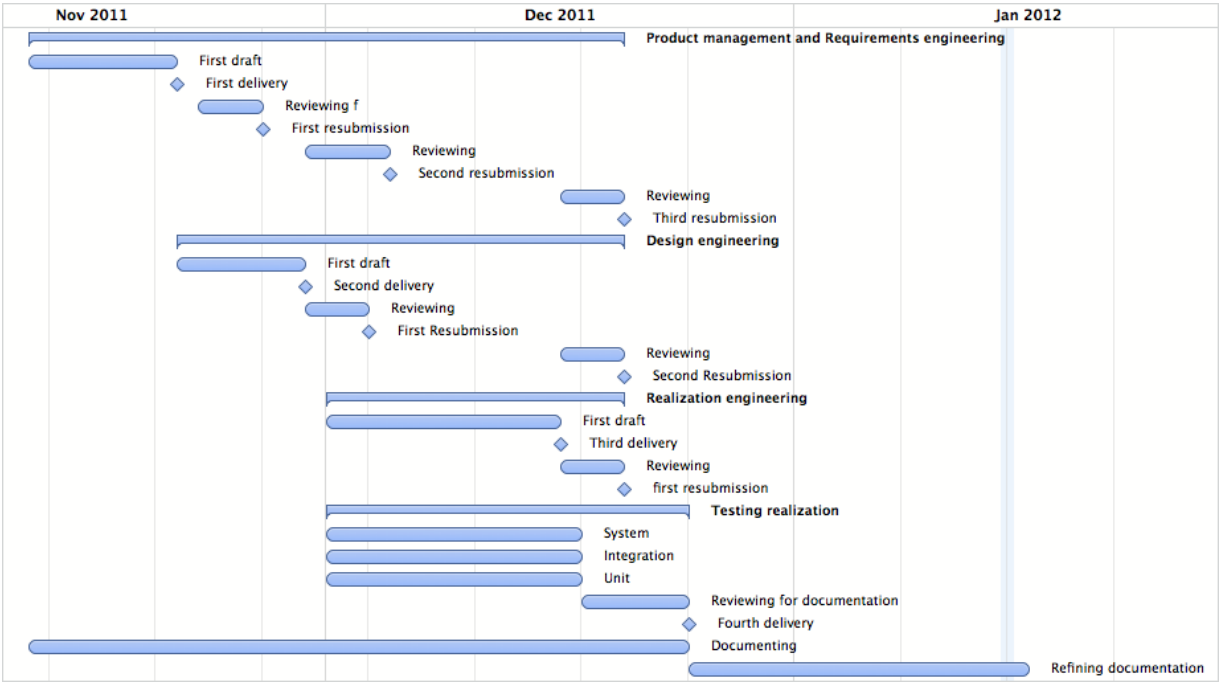


Figure 6.2: Web-line project plan.

As showed by the chart above, all the sub-processes artifacts have been delivered (according to the different phases) and reviewed iteratively until the final testing phase. As described by the SPLE

framework (in *testing engineering*), activities should re-start from product management, even though it was not applied in our project.

6.2 Risk Management

In order to reduce the risks that can lead the project to fail, risk management has been conducted following these seven guidelines we took from [18] :

- Open communications: informal discussions were conducted before, during and after each assigned task.
- Integrated management: risk identification, analysis and planning were defined process by process in verbal fashion since the amount of time assigned to each milestone has been planned weekly during the life-cycle.
- Teamwork: since the success of any software lies on mutual understanding of conflicting objectives, to mitigate exposure to this risk, *voting* for designating a project manager of each sub-process was iteratively conducted.
- Continuous process: to each sub-processes stand-up meetings have been conducted to maintain active discussions and at the same time surveillance on defined project risks as they change and evolve over time.
- Forward-looking view: to keep the focus on the same objectives, the team-members were not divided between domain and application engineering. Moreover sub-processes activities were assigned in equal proportion and discussed entirely from all the team-member (i.e. cross-functional approach).
- Global perspective: team-members had the same project goals definition since they were clearly stated since the first delivery milestone.
- Shared product vision: shared knowledge were discussed to achieve a clear vision of their functionalities since the team-members had to define two families of products with different capabilities. Changes has been conducted, properly tracked and pointed out in the following meetings.

6.3 Sustainment Engineering: Product-line Evolution

Software sustainment is “the processes, procedures, people, materiel, and information required to support, maintain and operate the software aspects of a system” [19].

SPLE involves the development of two main software aspects: domain and application engineering. Therefore, management for the scalability of *Web-line* aimed to combine the development of the SPLE platform with the development of the products, minimizing problems during the evolution of the project. Coordinating the activities of domain and application engineering required the definition of an internal integration processes consisting in the following activities: maintenance of the platform, bug fixing and deploy, custom enhancements, acting as feedback to platform and continuous integration.

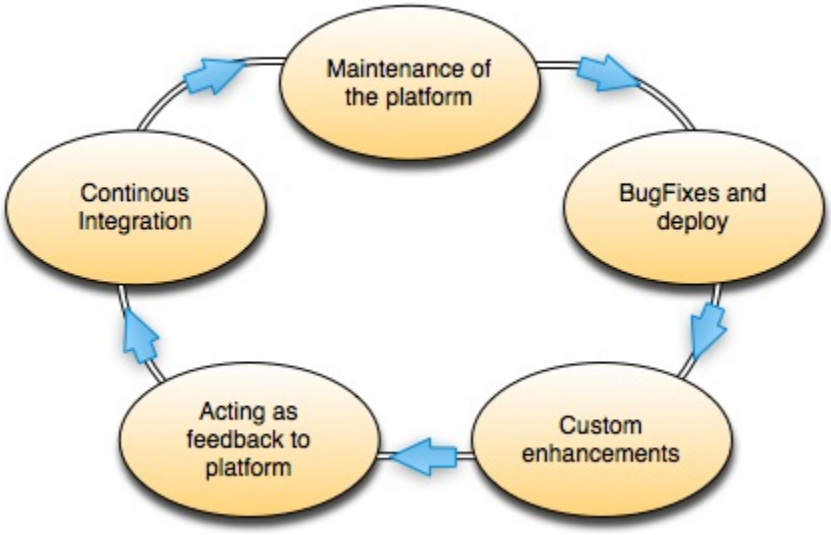


Figure 6.2: Integration management within the Web-line evolution-process.

We defined the maintenance of the platform as the first activity of the evolution-process: modifications have been tested by adapting the platform code to *stubs* of custom enhancements for development integration. Thanks to system testing, potential bugs were fixed and the changed code were tracked and deployed by the team-member in charge of the task. Then, custom enhancements derived from application engineering were implemented and substituted to the defined stubs. Therefore, the feedbacks from the final products needed to be validated to re-analyze the product-line platform. At the end, with continuous integration, all the artifacts of both domain and application engineering sub-processes were consistently combined. Briefly this was the process of management, that was iteratively conducted through all the life-cycle of the domain and application engineering.

7. Data Collection, metrics and tracking

To support management decisions, measurements have been necessary. By tracking and analyzing relevant attributes of the processes and obtained products as metrics, it was possible to provide a window toward the overall sub-processes development. The measurement activities involved two phases: initiation phase for designing the collection of data and performance phase for analysing the derived results. Within the initiation phase, we had to designate the goals, defined as follows:

- Improve performance in terms of effort spent for product development
- Reduce costs in terms of effort to change or introduce new plug-ins per product
- Increase percentage of reused components among different products

Afterwards, we defined four metrics for tracking the progress-status toward those goals:

- *M1*: Non-cumulative expenditures per delivery (i.e. the first milestone of committed artifacts during each sub-process) of our team members, in terms of effort, , from project inception to project end (see *Fig. 6.2*).
- *M2*: Cumulative expenditures of our team members, in terms of effort, in introducing one more plug-in .
- *M3*: Cumulative expenditures of our team members, in terms of effort, in changing one plug-

in.

- *M4*: Percentage of reused components among the different products.

Then we defined the data that must be collected in order to assess those goals:

- Time spent in analysis, design, development and testing of the platform and three specific products from each team member; time spent in developing the entire platform and three specific products from each team member per delivery.
- Time spent for introducing into the SPLE new plugins: *P&S*, *JobOpp, Portfolio* (defined into the reference architecture); time for maintaining SPLE to get it ready for working with the new plug-ins.
- Average time spent for changing one feature of three plugins, which already exist in the SPLE.
- Average number of LOCs of shared files among ten different products.

Finally, we reported the obtained results and related issues. From the metric *M1* the effort spent in development at different stages shows the prominent difference of man-hours spent for the product-line platform in comparison to the effort spent for the three different products (random sample), revealing lower total cost production, as expected. Therefore, even though the report shows the up-front effort (introduced in the SPLE overview section) spent for the platform development, the sample of derived products clearly present achieved pay-off.

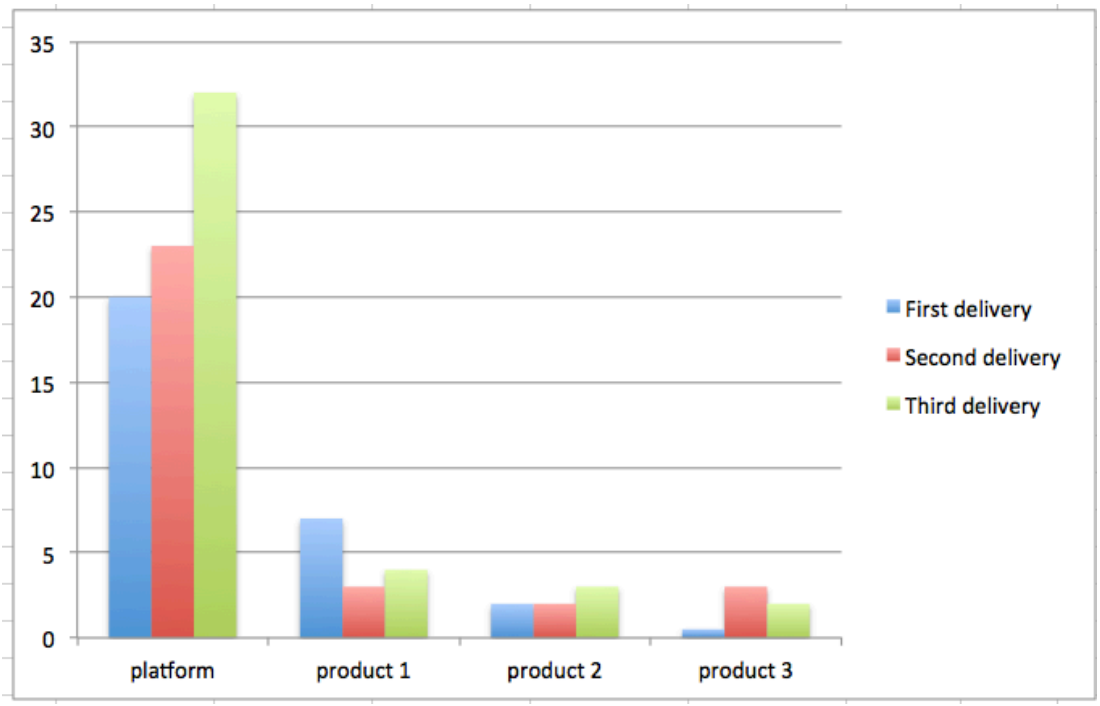


Figure 7.1: Effort (hours/person) in development per delivery

The same statistics can be shown from the development life-cycle perspective. All the phases of the platform development show much more effort spent in comparison to the specific products. The advantage is visible when the creation of product 2 and product 3 were more efficient, since most of the features from analysis to testing were almost available.

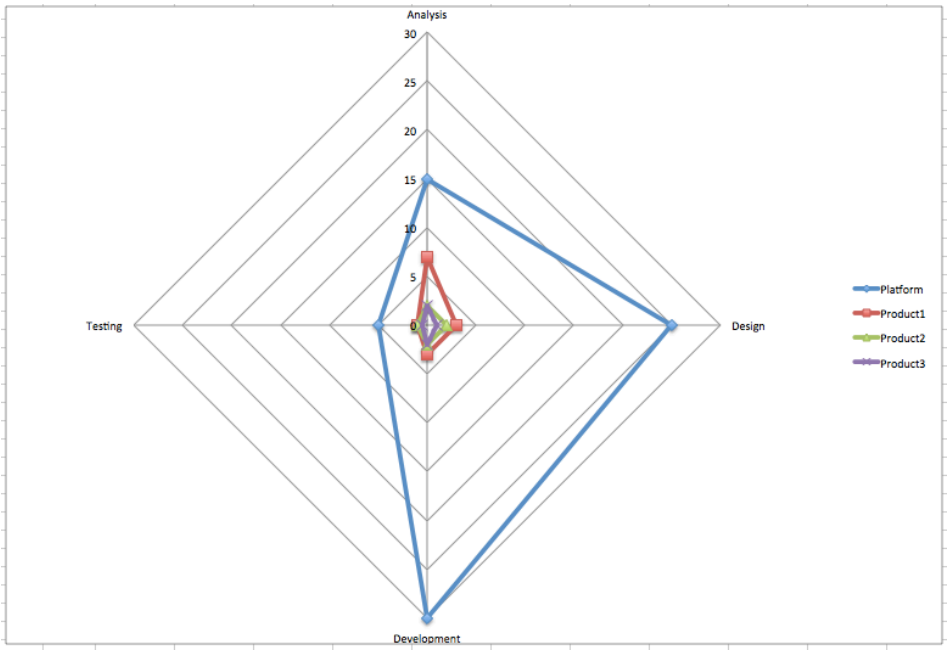


Figure 7.2: Effort (hours/person) per life-cycle phases

Introducing the second metric *M2*, we can assess the effectiveness of the design patterns presented in the reference architecture. The capability of adding new plug-ins, associated to the evolvability quality attribute, has been examined implementing the black-board architectural style. The following figure compares the development of three plug-ins with the effort spent for integrating and maintaining them in the SPLE. Proportionally, it is evident that the main challenge in introducing new features is referred to the development of the plug-ins themselves: only few changes are required in the product-line code (for more details refer to domain realization).

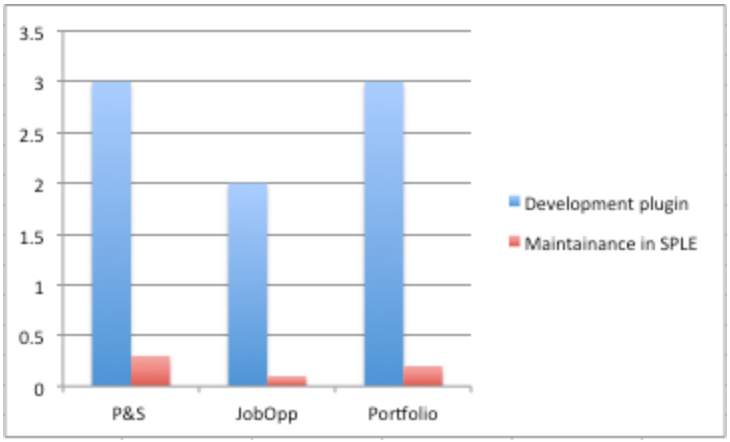


Figure 7.3: Effort (hours/person) per introduced plug-in

Then, from the third metric *M3*, it is possible to measure the total effort of *corrective* maintenance of a plug-in compared to the maintenance of the plug-in itself, and the changes needed in the SPLE. From the figure below it is possible to notice that the main time-consuming activity is related to the plug-in code, discussed in the previous situation as well.

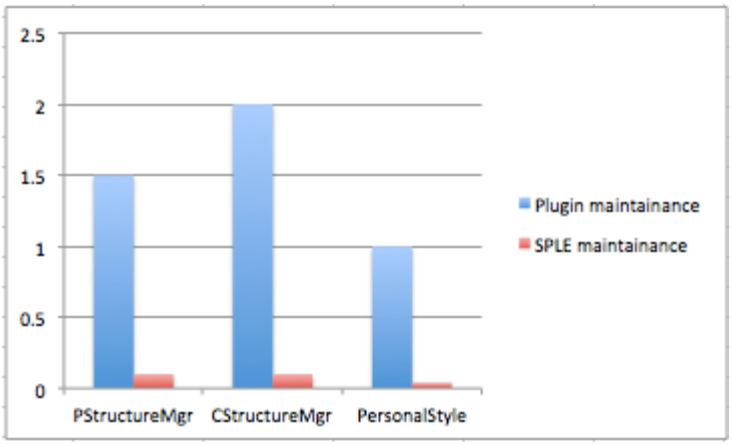


Figure 7.4: Effort (hours/person) per changed plug-in

The last metric *M4*, shows the measure in percentage of the code referred to common artifacts and variability in ten different products. Moreover, reused code of SPLE has been taken into account in order to highlight the variability already implemented in addition to the commonalities.

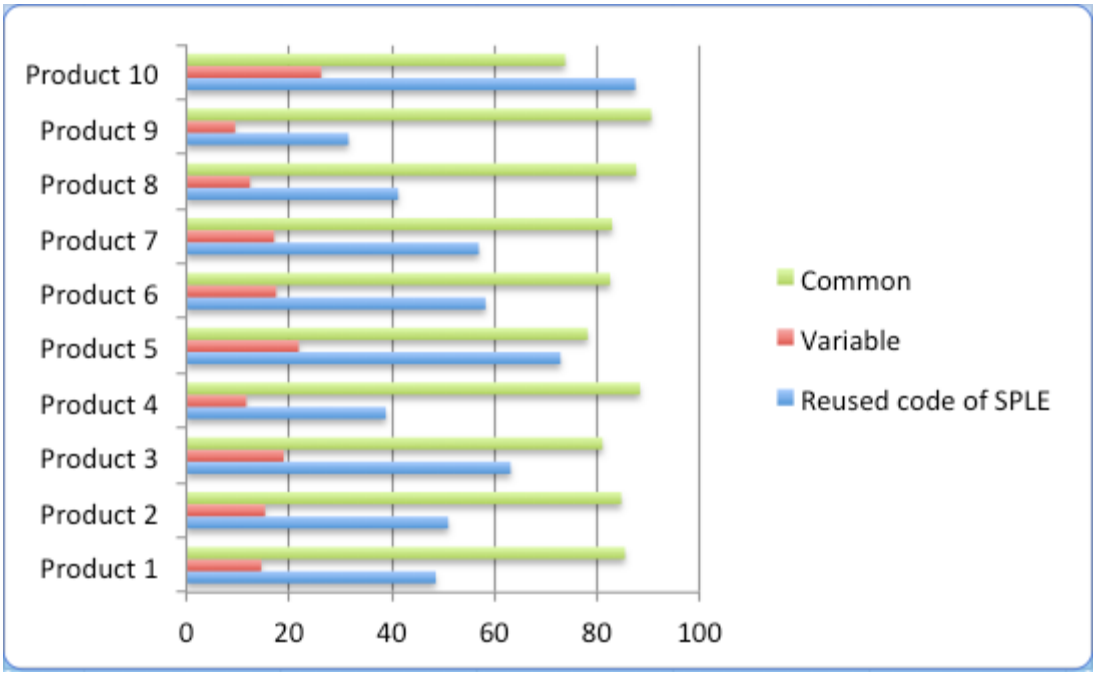


Figure 7.5: LOCs in percentage of reused, variable and common components

The total size of produced artifacts is presented in the next figure classifying LOCs by the different programming language.

Language	files	blank	comment	code
Java	41	835	425	2094
Javascript	2	86	41	896
CSS	8	291	423	462
HTML	14	91	113	306
JSP	3	105	2	271
XML	3	27	14	88
SUM:	71	1435	1018	4117

Fig. 7.6: LOCs overview per language

All the data have been collected using tools for *Eclipse* : *subversive* and *eclipse metrics* plug-in. Within the performance phase, we have collected the specified data and analyzed them by comparison with the expected advantages discussed in the overview section of SPLE. During the life-cycle development we found the following issues:

- Time pressure during the third delivery during the development phase see (*Fig 7.1)*. We remedied to it increasing reused code in product-line development (*Fig 7.5)*.
- Low percentage of reused code when new products are introduced (in view of application-specific variants). We mitigated it by integrating application-specific variants in SPLE since they resulted to be useful for other products.

Nevertheless, we achieved most of the objectives and all the positive experiences are going to be detailed in the next section.

8. Experience and Future Research

In this section experiences and future research are presented for increasing the understanding of the educational lessons learned from this study and building positive and negative reflections for the prospective practitioners.

8.1 Practices

The first step was the analysis phase. In addition to the common requirements, following the product line approach, our team had to take into account the variability of the product line, which is an unique challenge of requirements engineering in this subject. Defined the two product families from the background and scope analysis, all the specifications have been retrieved in according to general needs of personal and companies web-sites.

This first task has been valuated of medium complexity from the team members in view of the unusual mission of our project. We started allowing all the possible configurations to the costumer, but when we arrived to the complexity of the variability model we realized to better define the common specification and afterwards deriving the requirements variability from the variability analysis. The challenge has led to learn how to exploit the potentiality of core assets definition [14] and management in a product-line.

Next step was the design phase. We tried to shape the product according to the following non-functional requirements: usability, portability and maintainability. In addition to that, the state of the art of product line has also suggested to focus on flexibility and evolvability properties. The latter quality attributes have represented a real challenge in view of the prominent variability aspects of our project. After discussions about how to address architectural solutions, the complexity of flexibility and evolvability qualities has led to introduce standard styles and pattern such as: pipe-lines, plug-ins and black-board patterns. All of them have been arranged in such a way that the easiness of developing would not have been compromised. Due to the optimized solutions from the customers' and developers' perspective this task has been valuated of high complexity.

Despite the design engineering delivery took more time then expected, it facilitated the development phase. For this task we took advantage of the components based architecture and the detailed design

specified in the realization delivery. In the domain realization all the reference architecture has been implemented. Then, the plug-ins have been bound in according to the variability models constraints in the application realization.

The team has valuated this task of medium complexity since all the specifications and design activities have been defined in details: challenges have been encountered for creating an user-friendly configuration mechanism to realize the different products. On one hand, the easiness of configuring the variability of the different products, according to the specifications, allowed the use of the product-line to non-experts as well. On the other hand, to proceed with the configurations of the web-site contents, experts in html language need to be in place. At the end final users of *Web-line* can take advantage of products adapted to their needs and wishes with reasonable price since the product line engineering has helped to reduce the *Web-line* total production costs (effort).

All the produced artifacts have followed the sub-process activities described in the software product line engineering framework [3]. Through the development of each phase we had the possibility to return the previous stages, performing adjustments and improvements. Among the different feedback cycles, the one to highlight was between the design and requirements engineering, when the feasibility and characterization of variability models has been put in practice. Precisely the variability model has been revisited several times to correct the designed features expected from the requirements.

The essential activities of management, core asset and product development have been realized in continuous integration. As described the management has been kept simple in view of the small number of participants. The core asset development has carried out two product families with defined commonalities and production plan (defined by [17]) based on two months of scheduled time. At this regard the amount of time granted from the Prof. Ludwik K. has been valuated well-planned from all the team-members.

8.2 Success Factors

As remarkable success factors we highlight:

- Management to orchestrate and coordinate the interactions between domain and application engineering.
- Other knowledge applied from software engineering practices, technical and organizational management were revealed fundamental for the success of the project.
- Experience in the domain: useful knowledge gathered from expertise in the area of web-pages development.
- Early architecture focus: trying to foresee the degree of flexibility and reaching a proper trade-off between the amount of resources at disposal and the effort needed for the design.
- Platform development as first achievement: a good platform has led to increase the amount of code among the different products.
- Flexible and cross-unit team work: all the decision-making processes have been equally shared among the three work members; the roles between the overall software life-cycle have been equally distributed.
- Adequate supporting tools based on component development and object oriented paradigm have reduced the amount of required effort: components have been developed, compiled, linked and loaded separately dividing easily the work among the team-members; and object oriented paradigm enabled the encapsulation of information for naturally realizing and managing variability.

8.3 Benefits vs. Costs

The benefits of a platform in place have led to many advantages described during this study, but costs for maintaining it have not been presented exhaustively. The main challenge was related to the changes in the common requirements. They are high time-consuming and require group thinking because the goal usually was to arrive at the best solution for the whole family of the products. SPLE endows flexibility in solutions (see Domain Design section), but they are limited since not every change for one customer can be quickly achieved unless the experiences in the domain were deep enough to foresee the right variations.

8.4 Future Research

The perspective of developing the described practices in an hypothetical future project lies on the degree of interdependences of functionalities that will be defined among the different products. Despite the successful experience of *Web-line* experiment, interesting findings might arise from the real-world industries where customers are more demanding than in a simulated and controlled environment. Indeed, static web-sites are for their nature limited to a certain number of possibilities. In the future it would be interesting to explore and design product-lines able to assemble customized dynamic web-applications (see [20] and [21]).

During the two months of implementation of this project, the uncertainty of the products' analysis has been controlled by frequent discussions among the three team-members. In our case the small number of participants has led to many benefits in terms of decision-making.

All the procedures for performing changes and adaptions were anticipated from informal discussions. When we reached a compromise, all the artifacts modifications were informally reported. This approach could not be feasible if the amount of analysts, designers and developers increases. If that is the case, management processes could be put in place for tracing all the taken decisions (agreed and rejected). Moreover, instead of completely rely on up-front defined stakeholders' needs, positive feedback should be considered during all the product line life-cycle (by making it more *Agile*). This practice might increase the chances of success, reducing risk through prototyping and defining requirements in an evolutionary fashion [12].

9. Conclusions

In this document we tried to approach Software Product-line Engineering in a *student-friendly,* pragmatic and practical way, different from the traditional reference-books. We reported our concrete experience in the subject trying to give a detailed explanation of all the processes and sub-processes related to product-line engineering. In order to guide and facilitate the reader during the learning process we illustrated the case-study of our product-line project, *Web-line*, with as many significant examples as possible.

We started our *Web-line* project with the main objective of building a product-line that could save

precious time and effort to developers in the process of creating a new website. We finally achieve what we wanted and proved that spending time in designing a product-line, in our specific context, was much more convenient that building several similar websites from scratch. Indeed we were able to reach the *break-even* point quite fast: we estimated that the effort necessary to build the whole *Web-line* is comparable to the effort necessary to build *three* single applications from scratch.

In this paper we tried to cover relevant aspects of software product-line while recording our effort in building our *product-line*. Our main purpose is to transmit a set of practical *learned lesson* and somehow fill the gap caused by a lack of practical examples in the literature. We hope you find it useful.

Carmine, Farnaz and Nicolò - Blekinge Institute of Technology, 2012.

10. Bibliography

1. F. P. Brooks, "No Silver Bullet Essence and Accidents of Software Engineering," *Computer* 20, no. 4 (April 1987): 10-19.

2. JS Poulin, *Measuring Software Reuse: Principles, Practices, and Economic Models* (Addison--Wesley, 1997).

3. Klaus Pohl, Günter Böckle, and Frank van der Linden, *Software product line engineering: foundations, principles, and techniques* (Birkhäuser, 2005).

4. Mark A. Ardis and David M. Weiss, "Defining families: the commonality analysis (tutorial)," in *Proceedings of the 19th international conference on Software engineering*, ICSE '97 (New York, NY, USA: ACM, 1997), 649–650.

5. "Site map - Wikipedia, the free encyclopedia". Available: http://en.wikipedia.org/wiki/Site_map . *[Accessed: 10-Jan-2012]*

6. *"App Engine Java Overview - Google App Engine, Google Code". Available: http://code.google.com/appengine/docs/java/overview.html . [Accessed: 10-Jan-2012]*

7. M. Hewett and R. Hewett, "A language and architecture for efficient blackboard systems," in *, Ninth Conference on Artificial Intelligence for Applications, 1993. Proceedings* , 34-40.

8. Felix Bachmann et al., "A Meta-model for Representing Variability in Product Family Development," in *Software Product-Family Engineering*, ed. Frank J. Linden, vol. 3014 (Berlin, Heidelberg: Springer Berlin Heidelberg, 2004), 66-80.

9. Daniel Hoffman, Paul Strooper, and Lee White, "Boundary values and automated component testing," *Software Testing, Verification and Reliability* 9, no. 1 (March 1, 1999): 3-26

10. Verdin, K.C. et al., Definition of a Software Product Line Portfolio Using the Kano Model. *Science*.

11. Roos-frantz, F. et al., Automated Analysis of Orthogonal Variability Models . A First Step. *Structure*.

12. K. Mohan, B. Ramesh, and V. Sugumaran, "Integrating Software Product Line Engineering and Agile Development," *IEEE Software* 27, no. 3 (June 2010): 48-55.

13. B. Boehm, "Managing software productivity and reuse," *Computer* 32, no. 9 (September 1999): 111-113.

14. Paul Clements and Linda Northrop, *Software product lines: practices and patterns* (Addison-Wesley, 2002).

15. Web-line Project , hosted on Google Projects - Available: http://code.google.com/p/web-product---line/. *[Accessed: 15-Jan-2012]*

16. Web-line (beta), deployed on Appspot - Available: http://productlinearchitecture.appspot.com/ . *[Accessed: 15-Jan-2012]*

17. Ludwik Kuzniarz ,*PA2401 : Product-line Architecture*, Blekinge Institute of Technology,Winter Semester 2011.

18. Software Engineering Institute, *A Framework for Software Product Line Practice*, *Version 5.0* , Carnegie Mellon University,2009 , http://www.sei.cmu.edu/productlines/frame_report/org_risk_man.htm

19. Mary Ann Lapham and Carol Woody, *Sustaining Software-Intensive Systems*, May 2006, http:// stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA453287.

20. Karam, M. et al., 2008. A product-line architecture for web service-based visual composition of web applications. *Journal of Systems and Software*, 81(6), pp.855-867.

21. Balzerani, L. et al., 2005. A product line architecture for web applications. In *Proceedings of the 2005 ACM symposium on Applied computing*. ACM, pp. 1689–1693.

22. Pohl Klaus, "The three dimensions of requirements engineering: A framework and its applications," *Information Systems* 19, no. 3 (April 1994): 243-258.