

# Conceptualization of Feature Models for multi-client capable mobile Applications

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Software Engineering & Internet Computing**

eingereicht von

**Philip Messlehner**

Matrikelnummer 0728061

an der

Fakultät für Informatik der Technischen Universität Wien

Betreuung

Betreuer: ao.Univ.-Prof. Dr. Christian Huemer

Mitwirkung: Dr. Christian Pichler

Wien, 10.10.2013

\_\_\_\_\_  
(Unterschrift Verfasser)

\_\_\_\_\_  
(Unterschrift Betreuer)



Philip Messlehner, Inzersdorfer Straße 107/8/9, 1100 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 10.10.2013

---

(Unterschrift Verfasser)

## **Abstract**

Nowadays, an increasing amount of data is stored in the cloud. At the same time, companies start using mobile applications to operate and improve their business processes. However, mobile applications offered by the cloud-storage providers do not satisfy the needs of such companies.

The following three problems have been identified which prevent the usage of these applications. First of all, the feature set of these applications is definite. Therefore, companies are not able to extend the applications to meet their requirements to handle their business processes. In addition, the applications do not enable the companies to apply their corporate identity on the user interface. And, finally, companies want to have control over the distribution of their applications and decide which users are allowed to download and use their applications.

The different kinds of variability encountered result from the various requirements of each company. The solution for these problems is to systematically deal with variability in such applications. Therefore, this thesis applies concepts from product line engineering to the domain of mobile applications. Hence, this thesis defines a feature model corresponding to the users requirements. Furthermore, concepts and techniques have been developed for implementing variability mechanisms in Objective-C. Consequently, a software product line has been created using these developed tools and the corresponding feature model. This allows the efficient derivation of customized products based on the software product line.

The resulting approach based on software product lines allows to create a flexible application which is extendable with custom modules and components requested by users. Through this customization and extension the application's features are adapted to the company's business processes.

## Kurzfassung

Die Datenspeicherung wird zunehmend in die Cloud verlagert. Gleichzeitig bauen viele Unternehmen auf mobile Applikationen, um Ihre Geschäftsprozesse abzubilden und abzuwickeln. Die bereitgestellten mobilen Applikationen der Cloud-Anbieter erfüllen jedoch nicht die Anforderungen dieser Unternehmen.

Dabei verhindern folgende drei Probleme den Einsatz bestehender Applikationen. An erster Stelle sind die Applikationen nicht um die notwendigen Funktionen erweiterbar die benötigt werden, um diese sinnvoll im Geschäftsumfeld einzusetzen. Des Weiteren gibt es keine Anpassungsmöglichkeiten, um die Benutzeroberfläche an die Corporate Identity einer Firma anzupassen. Der Vertrieb dieser Applikationen wird über AppStores abgewickelt, die Firmen selbst können somit auch nicht steuern, welche BenutzerInnen diese Applikationen nutzen.

Die Lösung für diese Probleme ist ein systematischer Umgang mit Variabilität in Applikationen, welche durch die unterschiedlichen Anforderungen der Unternehmen entstehen. Hierbei sollen Konzepte der Produktlinienentwicklung in die Domäne der mobilen Applikationen übertragen werden. Aus diesem Grund wird in dieser Arbeit ein Feature Modell erarbeitet, welches die Anforderungen der Unternehmen widerspiegelt. Des weiteren werden Mechanismen entwickelt, um die unterschiedlichen Arten von Variabilität in Objective-C abzubilden. Sowohl das Feature Modell als auch die entwickelten Werkzeuge werden verwendet, um eine mobile Software-Produktlinie zu schaffen. Von dieser Software-Produktlinie sollen die verschiedenen Applikationen für Unternehmen effektiv abgeleitet werden.

Mit der entwickelten Software-Produktlinie ist es möglich flexible Applikation abzuleiten, die um Module und Komponenten je nach Wunsch der KundInnen erweitert werden können. Somit können die Applikationen auch auf die jeweiligen Geschäftsprozesse perfekt zugeschnitten werden.

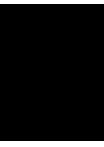


# Contents

<b>Abstract</b>	<b>ii</b>
<b>Kurzfassung</b>	<b>iii</b>
<b>Contents</b>	<b>v</b>
<b>1 Introduction and Motivation</b>	<b>1</b>
1.1 Problem Statement . . . . .	2
1.2 Outline of this Thesis . . . . .	3
<b>2 Requirements Analysis</b>	<b>5</b>
2.1 File Syncing and File Browsing . . . . .	5
2.2 Meta Information Synchronization and Manipulation . . . . .	8
2.3 File Management and File Organization . . . . .	9
2.4 Security . . . . .	10
2.5 Communication Aspects . . . . .	11
2.6 Customizable User Interface . . . . .	11
2.7 Custom Modifications . . . . .	12
2.8 Analytics Services . . . . .	12
2.9 Non-Functional Requirements . . . . .	12
2.10 Overview of Requirements . . . . .	14
<b>3 Related Work</b>	<b>17</b>
3.1 Software Product Lines . . . . .	17
3.2 Variability in Software Product Lines . . . . .	23
3.3 Variability Modeling Techniques . . . . .	28
3.4 Software Product Lines and Variability Mechanisms in Mobile Context . . . . .	38
<b>4 Objective-C Principles</b>	<b>41</b>
4.1 Key-Value Coding (KVC) . . . . .	41
4.2 Objective-C Runtime . . . . .	43
4.3 Class Extension with Categories . . . . .	45
4.4 Notifications . . . . .	46
4.5 UIAppearance . . . . .	49

<b>5</b>	<b>Implementing Variability in Objective-C</b>	<b>51</b>
5.1	Inheritance . . . . .	51
5.2	Parameters and Configuration . . . . .	55
5.3	Interceptors . . . . .	57
5.4	UIAppearance Wrapper . . . . .	63
<b>6</b>	<b>Variability Model for multi-client capable Mobile Applications</b>	<b>65</b>
6.1	Feature Model of the Mobile Application . . . . .	65
6.2	Mapping Features and Variability Mechanisms . . . . .	70
6.3	Realization of the Mapping . . . . .	71
6.4	Configuration Management . . . . .	80
<b>7</b>	<b>Evaluation and Discussion</b>	<b>83</b>
7.1	Product Derivation . . . . .	84
7.2	AESchreder Austria . . . . .	85
7.3	Anonymous Furniture Department Store . . . . .	86
7.4	Mercedes Benz Austria - MBÖ App2Date . . . . .	89
7.5	Critical Discussion . . . . .	92
<b>8</b>	<b>Conclusion and Future Work</b>	<b>95</b>
8.1	General Observations . . . . .	96
8.2	Challenge of Mobile Applications . . . . .	96
8.3	Future Work . . . . .	97
	<b>Bibliography</b>	<b>99</b>





# Introduction and Motivation

*Cloud* became one of the most used words in the information technology area in the last couple of years. More and more people are storing their information and documents in the cloud.

Companies are moving their storage from their own servers to cloud services. For instance, Box<sup>1</sup> mentions on their website that over 180,000 companies are using their service, including about 97 % of the Fortune 500s<sup>2</sup>. Reasons for companies to store their data in the cloud include the following.

**Outsourcing.** By outsourcing parts of the IT infrastructure to a cloud service provider, costs can be reduced.

**Collaboration.** Cloud services often offer additional features besides the simple storage of files. Having documents that can be edited from multiple people at the same time is one of these features.

**Offline Availability.** When using traditional file storage approaches data will only be available when connected to the server. Cloud-based services offer different client software for multiple platforms to make documents available offline. These clients keep track of any changes and therefore the data is consistent all the time.

**Mobility.** As mentioned before, the offered client software is available on multiple platforms, including mobile devices. Through the ability of using a mobile client to access the documents in the cloud entire business processes can be changed and improved. Field workers can access their files on their tablets or smartphones when talking to new customers or present new products

---

<sup>1</sup><https://www.box.com>

<sup>2</sup>Fortune Global 500 is an annual ranking of the top 500 corporations in the world and published by the Fortune magazine

from within such a client software.

However, the mobile clients offered by the cloud storage providers often do not satisfy the special needs of the companies. The most important requirements of companies are described as follows.

**Custom User Interface.** The user interface should reflect the companies corporate identity. This should be kept in mind regardless of whether the companies are using the mobile clients from the sync providers just for internal communication inside the departments of the company, or if they are actually selling their products with this application by presenting the products from within their cloud storage applications (i.e. using the application as a sales tool). The application itself should look like a product offered by the company.

**Additional Features.** Sometimes companies do want their own features included in cloud storage applications. The applications built from the cloud storage providers are non-modifiable and non-extendable standard software. This negative aspect may cause that this application will not be used by the company, if a mandatory feature is not included in this standard application.

**Distribution.** Having an application available in an *AppStore* makes it available for everybody to download and install it on their device. Companies want to control which people are allowed to install and use their application for security matters. So even if the user interface and features of cloud applications would be customizable, companies want to have control over the distribution of this application.

Due to these facts, the only solution to satisfy a company which is using a cloud service on mobile devices is a dedicated enterprise application. This means that the application is not published in the *AppStore* but distributed from the company itself. The enterprise application should be designed with respect to the companies corporate identity including all the specific requirements to fit their business processes. To summarize, since these business processes vary from company to company, individual requirements are present. Therefore a standard software cannot be used to satisfy these companies.

## 1.1 Problem Statement

In this thesis the afore mentioned problems are addressed. In particular, this thesis addresses the following aims.

1. It will be determined which requirements related to such a product are requested by the users. These requirements will set up a feature set for such a product. It should be examined which features of this feature set are mandatory or optional and therefore differ among the products.

2. Another research area of this work is to investigate which variability mechanisms exist in software product line engineering. It should be investigated how these variability mechanisms can be implemented by using Objective-C. Moreover, useful tools should be generated on top of this implementation by using design patterns and object-oriented programming techniques. These tools should be applicable for every product line implemented with Objective-C.
3. The established requirements should be used to create a feature model for a software product line. This feature model should contain all necessary requirements from the requirements analysis and reflect the variability of such a product line. The established features should be mapped to the developed implementation concepts for representing variability.

The overall goal of this thesis is to use software product line engineering techniques to derive products represented as mobile applications with respect to the domain discussed before.

## 1.2 Outline of this Thesis

This thesis is organized as follows. In **Chapter 2**, requirements are collected for the chosen domain and grouped into functional and non-functional ones. The requirements cover aspects of file synchronization, file browsing, meta information synchronization and manipulation, file management and file organization, communication aspects, customizable user interface and other custom modifications. These requirements are collected from customer pitches and in form of informal interviews with project managers.

The principles of software product lines are discussed in **Chapter 3**. While examining the related work the terms and notations of software product line engineering are explained. The meaning of variability and the different types of variability are described in more detail in this chapter. Afterwards, modeling techniques are presented to express variability. Different variability mechanisms are introduced to explain how variability can be implemented in a software product line. To distinguish this thesis from other research, other papers in the area of software product lines with a mobile context are analyzed and discussed. In this analyzes it will be examined which aims this research is trying to satisfy and how this thesis differs.

In **Chapter 4**, basic concepts of the programming language Objective-C are introduced to create a knowledge base for the following chapters. The covered topics discuss the Objective-C runtime and how this tool can be used to collect information about loaded classes and methods. An overview about *Notifications* is given, which are providing a loosely coupled alternative to the well-known observer pattern. The *UIAppearance API* is presented to show how to configure the appearance of user interface elements of an iOS application efficiently.

In **Chapter 5**, variability mechanism examined in Chapter 3 are implemented with Objective-C by combining them with object-oriented programming techniques and design patterns. This combination of existing variability mechanisms, object-oriented programming techniques, design patterns and the ability of Objective-C is used to create new tools to establish variability

in a mobile application developed with Objective-C. One of these tools allows to configure the subclassing variability mechanism for each product to easily exchange classes and parts of the application. Another tool is created to allow to set up parameters of classes from a configuration file in a flexible and dynamic way. An *UIAppearance* wrapper is presented to change the style of an application with an applied stylesheet. Interceptor implementations are introduced to include code into the application's life cycle. These implementation variants will be compared to each other. All these tools are used to implement the software product line using Objective-C.

The requirements established in Chapter 2 are mapped to features in **Chapter 6**. With these features a feature model of the software product line will be built and afterwards a mapping to variability mechanisms will be given. This mapping is explained in more detail to underline how the developed variability mechanism tools from Chapter 5 are helping to create products of the software product line. To show how the user interface of the application can be extended in a variable way an alternative concept is introduced to load independent modules into the application.

At the end of this chapter, an introduction to the configuration management is given to describe how the project was set up to allow derivation of products. This includes an explanation how the files are split up in the Xcode projects. It will be described which files are responsible for the configuration and how these configurations affect the products of the software product line. How to keep track of the different versions of the code base and the different products using source control management is also introduced in this chapter.

The design decisions made are evaluated in **Chapter 7**, where scenarios are chosen that should be implemented to show the weaknesses and strengths of the built architecture. Metrics are collected during this evaluation, e.g. the number of changes in the code base, the number of subclassed classes, the number of additional classes for a product, the number of resources for a product (like configuration files, images, icons, etc.) and the number of lines which had to be written for the stylesheet. These metrics will be used to state a critical discussion at the end of this chapter.

In **Chapter 8**, conclusions are drawn about software product line engineering in the mobile context as well as the applicability of software product line principles in the above mentioned domain. An outline of future work is also presented at the end of this chapter.

# Requirements Analysis

This chapter will give an overview on the requirements that should be fulfilled from a cloud storage-based application. These features will later be used in Chapter 6 to build a variability model for the software product line.

The requirements were collected from requirement specifications from customers who were requesting an application with cloud storage synchronization. The list of requirements was refined and complemented throughout informal interviews and meetings with project managers.

To simplify reading, the requirements are grouped into functional (Section 2.1 to Section 2.8) and non-functional requirements (Section 2.9). At the end of this chapter an overview of all requirements is provided in Section 2.10.

## 2.1 File Syncing and File Browsing

This section covers requirements that are related to a file synchronization process from a server to the application. It describes requirements that include the synchronization process in general as well as the behavior of this process during the application's life cycle. Another requirement that is covered in this section is the type of download mechanism that should be provided from the application to allow to view files and folders on the mobile device.

### 2.1.1 File- and Folder-Structure Syncing

The major feature of this application is to sync files from an existing sync service and browse through them. To view files, the application should be able to download them. The application should save all relevant information about the files and folders, which can be retrieved through the service.

Customers are probably already using a service for storing their files and are using this service to spread information inside the company. Therefore the application should have the ability to connect to several services and also provide an easy way to integrate new services, instead of forcing the customer to use a set of predefined ones.

Services for which a sync interface may be implemented are partly typical cloud-storage services, but also traditional file-protocols/services or even custom web services. A list of relevant services is described below:

- **Cloud-storage services:**

- **Box<sup>1</sup>:** Box is an online file sharing and content management service which hosts the user's files in the cloud. It focuses on business customers and has its strengths in the area of security, user roles and rights. It offers collaboration features like comments on files and folders, different access rights for folders, and tagging to enhance the organization of the content. Box provides sync clients for different operating systems, for desktop systems as well as for mobile systems, and offers a freemium model with 5 GB of free storage. A REST API for syncing is available together with SDKs for different operating systems, which offer a simpler way to sync files, but also reduce some capabilities such as controlling how to persist the retrieved data.
- **DropBox<sup>2</sup>:** DropBox focuses on private users instead of enterprise customers and it seems more intuitive and simpler to use compared to Box. It also offers a way to share folders but omits ways to comment, tag files or append other meta information such as file descriptions. It is not possible to give users different access to folders, like read or write access. They offer two different kinds of APIs, one called *Sync API*, a simple way to sync files like their desktop client, and the so-called *Core API*, which exploits all features and therefore is more complex and powerful.
- **ownCloud<sup>3</sup>:** ownCloud addresses the problems of enterprise customers regarding the minimal security Dropbox offers. They place their product as a better solution for enterprise customers compared to Dropbox and also offer hosting for calendars and contacts. They support two variants of their service, the first hosted with a monthly subscription fee per user with support and some other goodies. The other variant is a self hosted service, therefore they offer a install package to setup an own ownCloud server. To access ownCloud's data a web-interface, a provided API or a WebDAV interface can be used.

- **Traditional protocols and services**

- **File protocols:** A lot of companies use traditional file protocols to access their files on a shared volume. Samba<sup>4</sup> is a free software implementation of the SMB/CIFS networking protocol to provide file service for Microsoft Windows clients and runs on most Unix systems. Unfortunately there is no performant way to sync these files with an iOS framework. Socket programming has to be used to establish connections

---

<sup>1</sup><https://www.box.com>

<sup>2</sup><https://www.dropbox.com>

<sup>3</sup>ownCloud projects website with download link to self-hosted server <https://owncloud.org>, commercial project website <https://owncloud.com>

<sup>4</sup><http://www.samba.org>

- and data exchange. This socket programming on a mobile application may cause that the sync is not fast and reliable enough as required.
- **Microsoft Sharepoint<sup>5</sup>**: Microsoft's Sharepoint product offers a lot of functionality. Besides intranet and extranet sites it also provides a document and content management system. It provides an SDK and API for ASP.net and also an undocumented REST service to retrieve the same information. This lack of documentation makes it very hard to access Sharepoint from other systems.
  - **WebDAV<sup>6</sup> (Web Distributed Authoring and Versioning)**: WebDAV is an extension to HTTP to provide functionality to edit and manage files. It provides eTags for version control and supports extensions to add additional meta information to files.
- **Custom web services**: Sometimes users want to provide their own web service to supply access to their files, e.g. those hosted in a proprietary intranet system. Despite of technical details the application should be able to integrate such custom services.

### 2.1.2 Sync Behavior

The sync behavior describes how the information about the directory and file tree is synced to the application. It depends on the directory structure and the amount of files and the type of directory organization (depth versus breadth) which sync behavior leads to the best experience. Some sync services may have the ability to support different sync behaviors, therefore the customer should have the ability to choose one of the following:

- **Recursive**: A recursive sync is only requesting information about the actual viewed folder. If accessing a subfolder, the elements of this subfolder have to be requested. Therefore it is not applicable for offline-browsing and will lead to bad experience, if the directory organization is slim but deep.
- **Full**: A complete or full sync will fetch the whole directory tree and is therefore best applicable for offline-browsing. However if the directory organization is slim but deep and there is no method to retrieve the whole tree with a single call, this behavior could lead to a massive amount of requests to retrieve all necessary data.
- **Delta**: Delta sync services are consuming a timestamp or key, which stores information about the last successful sync process. The service is responding with a list of files and folders, which have been changed, created or deleted since the last sync. This sync behavior is therefore only responding with necessary information, which may lead to smaller response sizes.

---

<sup>5</sup><http://sharepoint.microsoft.com/>

<sup>6</sup><http://www.webdav.org>

### 2.1.3 File Download

The different file download behaviors describe how the content of files will be synced to the application. The usage of the different behaviors depends on the needs of offline availability of files.

- **On demand:** On demand behavior will download files when they are explicitly requested, e.g. in cases where the user tries to open the file inside the application.
- **Custom:** The custom behavior will allow users to set up rules, which will be applied to all subitems of a folder or a set of files selected by the user. This collection will be available offline, therefore the application will download content on the very first sync and after a file got changed. To manage such a collection a custom way to organize the files and folders has to be provided (see Section 2.3).
- **Replication:** This behavior will force the synchronization engine to download all files of the directory tree.

## 2.2 Meta Information Synchronization and Manipulation

Regarding the selection of the sync service, the users have additional requirements which cover meta data manipulation or at least representation. Such meta data may appear as tags, comments or as file description. Most of them depend on the meta data provided by the service and their sync engines itself. This meta data may help the user to manage their files inside the application in an additional way or use the information to collaborate on a file. Well known meta data concepts are described below.

### 2.2.1 Tagging

Tagging is a modern way to organize files and folders and offers a different approach than the hierarchical file browsing everybody is aware of since the introduction of the first graphical user interfaces for personal computers. A lot of people are using these technique to speed up their search behavior and include meta information to their files.

Therefore the syncing and the presentation of this meta information is required for a lot of users. If this feature is included, it should also change the search behavior to retrieve files matching a tag name.

### 2.2.2 Comments

Collaboration and working in teams spread over the entire world are essential for internationally acting companies. One way to encourage files to a subject of collaboration is to link discussions about the content to the file. Some services already allow comments on files and folders. To allow collaboration through comments and discussion, this information has to be synced to the mobile device and included into the graphical user interface. A possibility to add these kind of meta information from within the application should also be provided.



### 2.2.3 Additional meta information

Some services are providing additional meta information like flags, a reference to the creator of a file, file descriptions, etc. A flexible way to save this additional information should therefore be provided.

An example for such an additional information is represented by a property which determines if the file should be only visible in a special mode, such as a private mode. Users want to switch between a public and private mode to hide or show these marked files. This could be useful for presentations where people are involved who are not authorized to see these files.

## 2.3 File Management and File Organization

Users want to have supplementary ways to organize and search for files to improve their workflow or to save some time. This group of requirements describes several ways to gain this benefit by introducing different metaphors and strategies.

### 2.3.1 Favorites

Favorites became an intuitive way to bookmark content for later consumption or just to advance retrieval. Therefore users want to be able to mark files and even folders as favorites. It should be easy to mark or unmark a file as favorite and the special favorite folder should be quickly accessible in every part of the applications interface.

### 2.3.2 Custom Collections

Sometimes the favorite technique is insufficient when organizing a large amount of data. It only flags content, but does not provide any other way to manage the flagged content. A lot of users require a way to group these flagged items, they want to have different favorite groups, they want to have custom collections. They are asking for a way to create their own folders and mark files, which should be accessible within these custom folders but also will not change the actual place where the flagged files are persisted. This file should remain at its source, the custom collection should only provide a faster way to access it, like a shortcut or cross-reference. These custom collections should also be identified by a name and also contain other meta information, such as notes.

### 2.3.3 History

Opening a file that has already been opened by the same user before is a very common use case. Thus a history is a practicable way to find this information more quickly. The files inside this history should be sorted by its access date and the user should be able to clear the history or remove specific items from the history log.

### 2.3.4 Saved Search

A lot of personal computer operating systems offer a highly configurable and fine-grained search functionality, but in fact it takes some time to set up a very complex search query. In addition, it happens very often that users want to execute the same search query in the future - they are searching for the same kind of files again. Consequently it is useful to provide a possibility to save these search queries as intelligent folders to fulfill this requirement.

## 2.4 Security

When saving personal information within an app, security becomes a major fact of interest. Applications should treat files as properties of the users and therefore implement mechanisms to protect these properties.

### 2.4.1 Passcode

Passcode locks became very popular through the high prevalence rate of smart phones and tablets. This feature offers a low level security mechanism to prevent usage by unauthorized people. Not only devices (or actual mobile operating systems) itself use this way to raise security, but also some mobile applications like Dropbox<sup>7</sup> or 1Password<sup>8</sup> have introduced passcode locks to protect sensible data.

### 2.4.2 Authentication

For some services it is necessary to provide a way to authenticate users. The commonly used authentication methods in applications can be separated into two groups.

**Authentication Redirection.** Some of these services are using single sign on, which will interrupt the application workflow, redirect to a browser where the user has to enter his credentials and will redirect back to the application after success.

**In-App Authentication.** Some services allow to pass the credentials directly to the service endpoints (e.g. basic authentication). Hence the necessary information like username and password can be entered from within the application. This authentication mechanism also allows to use the same account for each user, like in a product catalog application for customers. Therefore a silent authentication in the background can be provided for this authentication mechanism.

---

<sup>7</sup><https://itunes.apple.com/us/app/dropbox/id327630330?mt=8>

<sup>8</sup><https://itunes.apple.com/us/app/1password/id568903335?mt=8>

## 2.5 Communication Aspects

*“More Collaboration, Less Frustration” – BOX<sup>9</sup>*

Collaboration has become more and more important throughout the last couple of years. Nearly everybody has to work in teams, for this reason communication is very important to achieve goals and success. This group of requirements describes ways to support these social and communication aspects.

### 2.5.1 Timeline

Since Facebook introduced its timeline<sup>10</sup>, this way to represent events has become a widely used metaphor. Some cloud storage services also introduced such a timeline to list the events involving the files of the users to supply a better overview. This timeline shows new or modified files, new comments from other users collaborating with your files or other file-related actions. If requested by the user and supported from the sync service, this feature should be included into the application as a separate module.

### 2.5.2 Notifications

Notifications of applications are used to establish a communication channel to the users. They are used to inform about important news and updates or include other important information. These notifications could also be used inside an enterprise application to spread company-based news within this company. An additional requirement is represented by the need of having an archive to access older notifications. Notifications combined with an archive provide an alternative way of a blog or news feed within an application and, due to this, improve the internal communication of a company when used inside an enterprise application.

## 2.6 Customizable User Interface

It is very important for companies to have their corporate identity on every document and even on every product they are giving to their customers. The fact that they also want to have their corporate identity applied on internal documents or information which are only shared within the company underlines the importance of these design guidelines. Hence applications that are used from the company should also be designed with respect to their guidelines which implies a corporate identity-driven design of these applications.

### 2.6.1 General User Interface Element Customization

Users want to apply themes on the application, i.e. their logo should be shown on every screen of the application and the controls should be colored with the company colors.

---

<sup>9</sup><https://www.box.com/business/project-collaboration/>

<sup>10</sup><https://www.facebook.com/about/timeline>

### **2.6.2 Custom User Interface Elements**

Sometimes applying a theme is not enough to reach satisfaction regarding the customization of the user interface and replacements of user interface elements like buttons or other controls have to be done to reach this goal of user interface customization.

## **2.7 Custom Modifications**

Occasionally users have totally different ideas how to extend an application and meet all their needs, especially when they are trying to improve their business processes. In these cases, heavy customization is necessary to satisfy these users.

### **2.7.1 Custom Modules**

There might be circumstances where a user wants to have a complete new module included into the application, in fact he probably wants an entirely independent module, which acts like a mini application inside the existing application or a module which uses the data synced and stored from the original application logic.

### **2.7.2 Custom Components**

It is sometimes necessary or economical to exchange small parts of an application with other parts, rather than extending or modifying them to meet special needs for a single user. An example for this problem would be the requirement to show a custom mail dialog instead of the built-in mail dialog from the iOS operating system.

## **2.8 Analytics Services**

To track the users of an application and analyze the usage of the different features of an application, analytics services can be used. The application should have the ability to integrate different analytics services depending on the needs and preferences of the customer to collect such information. This information may also be used to improve the application's user interaction and may show the usage frequency of certain features.

## **2.9 Non-Functional Requirements**

The following listed requirements are not related to a specific feature and represent non-functional requirements for the application. Nevertheless, they are as important as the previously mentioned requirements.

### 2.9.1 Data Protection

The application stores business relevant and sensible data. Therefore, the protection of this data is essential. One aspect is the visibility of the sensible data. No data should be retrievable from outside, therefore iTunes file sharing should be turned off<sup>11</sup>.

Another aspect is covered by the encryption of the files, when stored on the mobile device. A compromise between performance of file reading and encryption-policy of downloaded data has to be found.

### 2.9.2 User Experience

Animations and intuitive interaction concepts can help to improve the user experience. Current concepts used in iOS applications and mentioned in the several guidelines should be considered during implementation of the user interface and application workflow.

### 2.9.3 Performance

Bad performance can lead to a negative impact on the users' experience while using the app, especially when the interaction with the user interface is not performing well. Furthermore, it is very important to shift heavy work (like file synchronization and data storage) to the background to not interrupt the interaction.

### 2.9.4 Reliability

It is evident that the application should be well tested and should not crash very often during usage. In case of a crash, a report should be send to a server and the developer should be informed to fasten up the bug fixing process. Moreover, a way of giving feedback to the developer should be provided for future improvements.

### 2.9.5 Maintainability and Extensibility

The fact that the application is used with different configurations and different modules from various user groups increases the complexity of the system. It should be taken care of this increase of complexity during the whole implementation process. It is possible that a new user requests a totally new feature or wants to include a special module into the application, which does not exist yet. To allow the usage of custom component parts of the application have to be designed capsulated. Dependencies between such capsulated parts should be minimized to allow substitution of these components. This also leads to an improvement in the area of maintainability since the components should not greatly influence each other.

---

<sup>11</sup>iTunes file sharing allows applications to display their content of the *Documents* directory inside iTunes while the iOS device is connected. The user can drag files from and to a specific area to copy files into this directory, or remove files from this directory. For example, it could be used in a video player application to sync movie files to the application.

### **2.9.6 Update Mechanism**

To improve the procedure of bug fixing, but also the process of prototyping and beta testing, a system that allows over-the-air distribution of new versions of the application should be used, including a mechanism to force the user to update the version (e.g. for critical bug fixes). A system which provides analytics about the distribution of the different versions is desirable.

## **2.10 Overview of Requirements**

Table 2.1 shows a summary and an overview of all functional and non-functional requirements. The requirements in this table are structured and regrouped as a list to simplify the creation of a mapping from requirements to features and to variability mechanisms to implement these features. This mapping will be established and discussed in Section 6.2.

Functional Requirements	
<ul style="list-style-type: none"> <li>• File- and folder-structure syncing</li> <li>• File download on demand, replication or custom</li> <li>• Meta information support <ul style="list-style-type: none"> <li>– Tags</li> <li>– Comments</li> <li>– additional meta information</li> </ul> </li> <li>• Favorites</li> <li>• Custom collections</li> <li>• History</li> <li>• Saved search</li> </ul>	<ul style="list-style-type: none"> <li>• Passcode</li> <li>• Authentication</li> <li>• Timeline</li> <li>• Notifications</li> <li>• User interface customization</li> <li>• Custom user interface elements</li> <li>• Custom modules</li> <li>• Custom components</li> <li>• Analytics services</li> </ul>
Non-Functional Requirements	
<ul style="list-style-type: none"> <li>• Data protection</li> <li>• User experience</li> <li>• Performance</li> </ul>	<ul style="list-style-type: none"> <li>• Reliability</li> <li>• Maintainability and extensibility</li> <li>• Update mechanism</li> </ul>

Table 2.1: Overview of functional and non-functional requirements.





## Related Work

This chapter will cover fundamentals of software product lines to understand the underlying principles as well as to define the terminology. Subsequently, variability in software product lines will be examined in more detail, followed by an introduction to different modeling techniques to express variability of a product line. At the end of this chapter, other related papers will be briefly discussed to give an overview of scientific work in this area and to show the differences to the work presented in this thesis.

### 3.1 Software Product Lines

This section examines the ideas and terms behind software product lines and also mentions risks and benefits which come with them. In addition the processes of software product line engineering are discovered and described.

#### 3.1.1 Terminology

In [CN02] software product line is defined as follows:

*A software product line is a set of software-intensive systems sharing a common managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.*

The definition specifies that a software product line is composed of its products which share a set of features. These products are built upon core assets which represent the foundation of the software product line. These core assets can vary from product to product in a prescribed and controlled way. Core assets can be typified as code fragments, architectural specifications, documentation, domain models, requirement statements, test plans, process descriptions and other elements which help to develop multiple products. Building products with these core assets

is achieved in a managed way following predefined rules and processes to improve economic efficiency.

There are different terminologies used in scientific papers and books. This work will use the terminology of software product lines, core assets and products instead of product families, platform, or customization.

Taking this definition into mind, every product of the software product line is produced by taking core assets and optionally modify them in predefined variation mechanisms to match the use cases. If there is no core asset for a certain use case of the product, new core assets are added. Consequently, the build process of a product from a software product line can be seen as assembling and modification, not as creating per se. To make this assembling process more effective, software product lines should provide a guide that specifies this building approach.

### 3.1.2 Product Line Activities and Processes

The Software Engineering Institute (SEI, [Nor02]) defines three essential activities of software product line development. These activities are iterative and can be represented as rotating arrows. At the same time these activities get influenced by each other (see Figure 3.1) [Eic12].

Another way of separating the processes of a software product line can be found in literature [BKPS04][PBvdL05]. This separation defines two processes that are building the fundament of the software product line engineering paradigm – also mentioned as the software product line engineering framework. The aim of this separation is to focus on building of a set of reusable assets, as well as on the building of applications that satisfy the needs of the customers. The two processes and the influenced subprocesses are discussed below and are illustrated in Figure 3.2.

These two figures are connected as follows. *Core Asset Development* from Figure 3.1 is represented as *Domain Engineering* in Figure 3.2. *Product Development* from Figure 3.1 is represented as *Application Engineering* in Figure 3.2. The *Management* activity from Figure 3.1 contains sub-activities which can be assigned either to *Domain Engineering* or *Application Engineering*.

#### Domain Engineering

In this process, the commonality and variability of the software product line has to be defined. Therefore this process is responsible for creating the reusable assets consisting of the different artifacts for testing, requirements analysis, design, and development. Another key goal of this process is the definition of the scope of the software product line. It defines the set of products for which the software product line should be designed for. If this scope is too small, the core assets won't be built in the necessary generic way and therefore will not be eligible enough for future growth. Otherwise, if the scope is too large, the core assets will not be applicable for a huge set of products but instead only used in a few of them. Therefore, a lot of core assets have to be developed and the benefit of reusability will not be achieved [BKPS04]. Apart from the core assets and the definition of the scope, guidelines to build products from the software product line should be extracted.

This process gets influenced by the inputs that are described as follows. Product constraints define the possible variations between the products and their behavioral features as well as the

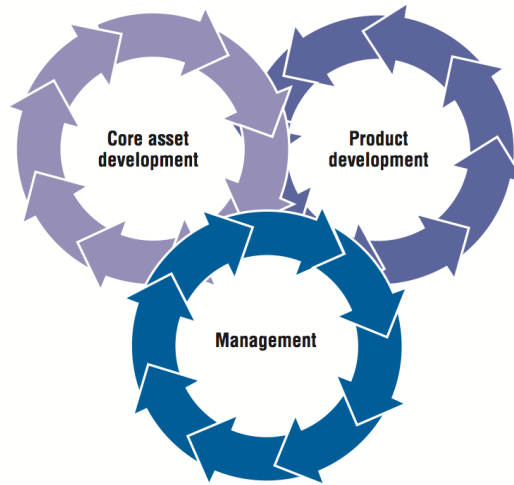


Figure 3.1: Essential product line activities [Nor02].

information which features will definitely be needed to meet the needs of the market or the customers in the future. Guidelines (style, patterns) and frameworks that are used to build core assets and products. Constraints which control the production, like company-specific standards or legacy restrictions. Such product constraints can also be represented as limitations regarding the infrastructure for production.

The domain engineering process consists of several subprocesses. The outcomes are several domain artifacts. **Product Management** deals with market strategy and economic aspects. **Domain Requirements Engineering** is responsible for documenting commonalities and variability of the system through requirements. Output of this subprocess is a variability model (see Chapter 6). **Domain Design** provides a high level structure for all products as a reference architecture. **Domain Realization** covers the detailed design and realization aspects of the reusable components. **Domain Testing** is necessary to verify and validate the reusable components.

### Application Engineering

Application Engineering represents the second process in the software product line engineering paradigm and is responsible for deriving a product from the product line, or to be precise, from the reusable assets. The process is exploiting the software product line's variability according to the needs of the product. Another task is the documentation of the application engineering artifacts. The guideline provided from the domain engineering process must be respected and satisfied. Similar to the first process, the application engineering process also consists of several subprocesses:

- **Application Requirements Engineering:** Focuses on the specification of the application with all its requirements. Deltas between these two requirement engineering subprocesses

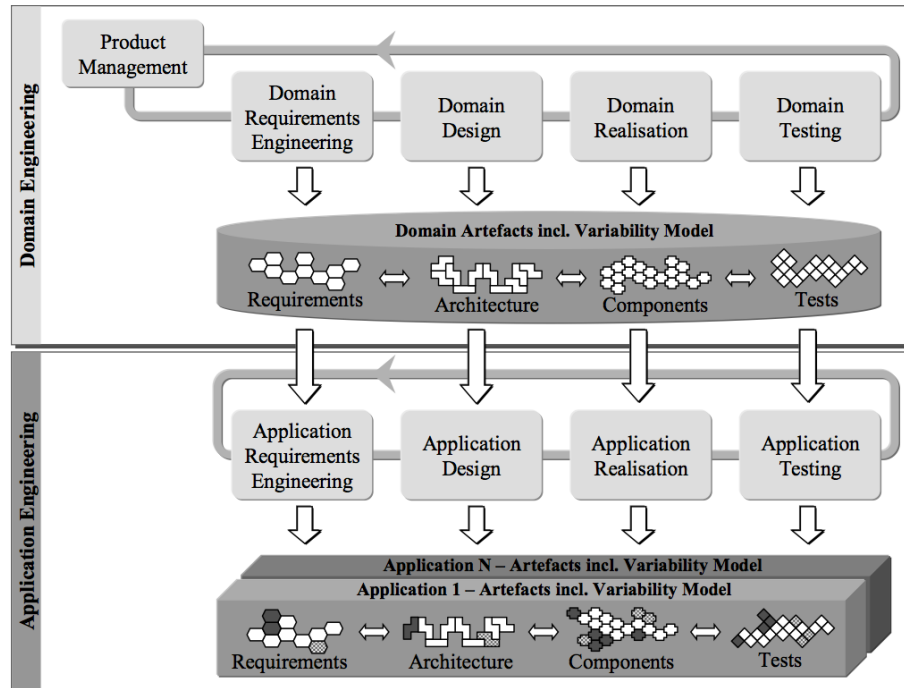


Figure 3.2: The software product line engineering framework [PBvdL05].

have to be evaluated afterwards to enhance future product developments of the software product line or to consider new assets for the domain.

- **Application Design:** Represents a specialization of the reference architecture and uses it as input. Differences can occur, which should also be examined carefully.
- **Application Realization:** This subprocess deals with the configuration and development of software components, components used from the domain engineering process.
- **Application Testing:** This last subprocess is responsible for the testing of the application including a validation and verification against the application's specification.

As mentioned before, the results of the several subprocesses should be evaluated afterwards to analyze the applicability of the domain assets and, as the case may be, also be reworked or refined. Therefore this process is acting as a feedback loop for the domain engineering phase to improve the software product line.

### 3.1.3 Motivation and Benefits

The definition in Section 3.1.1 describes the products of a software product lines as a software developed from a common set of core assets in a prescribed way. As mentioned before, this means that these core assets are reused strategically. This may cause a direct benefit each time

a new product of the software product line is built, regarding the reduction of the costs, saving time, improvements of software quality, and the reuse of resources and knowledge. How these benefits can be achieved is discussed in more detail below.

### **Reduction of Costs**

Reuse of assets implies cost reduction for each product. Hence reuse has to be planned beforehand to provide managed reuse, which requires an up-front investment to create the assets. Therefore the initial cost will be higher, but reduced after deriving more products (see Figure 3.3). The Break-Even point in this figure is reached after building three different systems [PBvdL05]. After this number of products, the costs for further systems will be significantly cheaper for software product lines than they are for one-at-a-time systems. This number is described in “It Takes Two” [CN02]. [CN02] also mentions several other papers which confirm this proposition. In [WL99] the authors claimed that product line engineering produces a payoff after about two to three systems. In addition [JGJ97] is analyzing the Break-Even point of reused components and marks this point with three to five reuses. This paper also reveals that the initial costs for such reusable components will be 1.5 to 3.0 times as much compared to similar components for single applications. They also claim that it takes two or three product cycles to get a significant benefit out of the reuse of these components. Several other cited works confirm the hypothesis of three produced systems (see [Rei97], [Pou97] and [Tra95]).

This will also lead to reduced costs for the customer who will not have to pay as much as they would pay for an individual software. Moreover, cost-estimation becomes simpler because the assets provide a sound basis for calculation. Only adoptions, new assets or extensions to meet the customers needs have to be calculated separately and developed from scratch. In general it is evident that the amount of time and costs spent at the beginning of the project will be much higher than in conventional projects. However, these additional costs will amortize after building more and more products. These additional costs comprise costs regarding requirements engineering and architecture design as well as test case planning and training of employees.

### **Time Saving**

Developing a software product line is time consuming, therefore it takes some time to finally ship such a system to the market. [PBvdL05] assumes, that this time is roughly constant for individual software products, but much shorter compared to the time the first products of a software product line will cause to be shipped. Figure 3.4 shows, that the time spent for building a product from the software product line decreases with the number of previously assembled products. This positive effect is an outcome of the reuse of artifacts in a software product line. Apart from that, individual software products will always cause constant time to be shippable.

### **Software Quality and Maintenance**

To test a software product line, generic test plans can be set up and applied to the products [CN02]. Only new components or modified and customized assets have to be tested in a more detailed way. This can also lead to a positive impact on the quality of the software product line, and

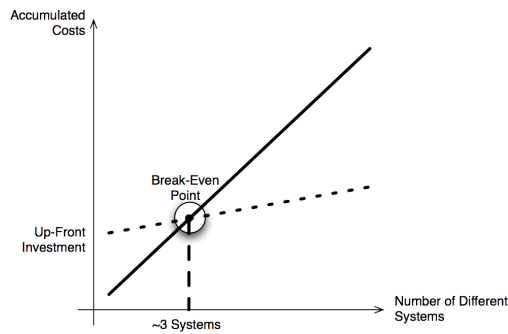


Figure 3.3: Costs for developing  $n$  kinds of systems as single systems compared to product line engineering [PBvdL05].

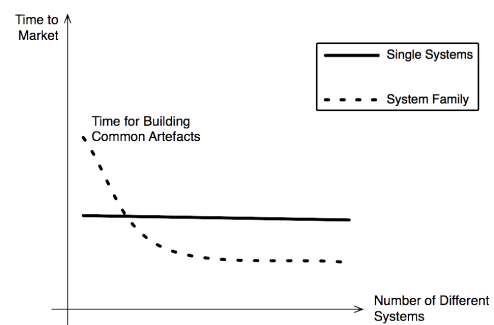


Figure 3.4: Time to ship a system to the market with and without product line engineering [PBvdL05].

therefore, for each product. Pushing the quality of one product to the next level (regarding performance, availability and so on) will cause an increase of quality for the whole software product line. Maintenance for products of a software product line is much easier than for multiple individual software products. When a problem gets encountered and fixed in a core asset of one single product of the software product line the same problem will be fixed in every other problem. This effect comes into being because the problem occurred in a specific asset (which is used in several products) and not the product itself.

### Reuse of Resources and Knowledge

Personnel can be easily transferred through projects covering different products of the software product line, because of the commonality among these products [CN02]. Their gained experience from one product can be applied to other products of the software product line as well. This also reduces the initial training when joining a team developing on a different product dramatically, which implies an increase of productivity. The effort that is necessary to train employees to use tools and processes has to be made only once, since the learned aspects can be applied to all products of the software product line.

## 3.2 Variability in Software Product Lines

Variability in software product lines is used to describe commonalities and variation among assets. This chapter will introduce some terms used in the literature as well as characteristics and classifications of variability. At the end, common mechanisms to implement variability will be discussed in more detail.

### 3.2.1 Terms and Notions

Spoken in a non-technical context, variability refers to the ability to change an object, especially the attributes of the property which are changeable and which values are valid. The following definitions are mentioned in [PBvdL05], where some helpful examples are given to support the understanding of these concepts.

#### Variability Subject and Variability Object

*A variability subject is a variable item of the real world or a variable property of such an item [PBvdL05].*

Derived from this definition a variability subject describes what subject is able to vary and helps to identify the variable item or property of the real world. The reason why such a subject is varying can have different reasons, for example different stakeholder or customer needs, different country regulations, technical reasons, etc.

*A variability object is a particular instance of a variability subject [PBvdL05].*

The variability object describes how such an subject can vary and numerates the possible shapes of the variability subject. As an example, a variability subject in the real world could represent a color property of items, whereas the different colors itself are representing the variability objects.

#### Variation Point and Variants

*A variation point is a representation of a variability subject within domain artifacts enriched by contextual information. A variant is a representation of a variability object within domain artifacts [PBvdL05].*

Core assets have to fit in more than one context to establish effective reuse. They are slightly changed or modified to fit in these contexts. The part of the application that occur in different shapes is called variation point. The possible values for such a variation point are called variants and therefore describe the ways an asset can occur in a specific variation point.

A really useful example is shown in Figure 3.5. The variation point in this example is represented by the color of a car whereas the instances or variants of this variation point are represented as a red and green car. Another example can be given with an asset which is responsible to persist data. The persistence type would represent the variation point of the asset,

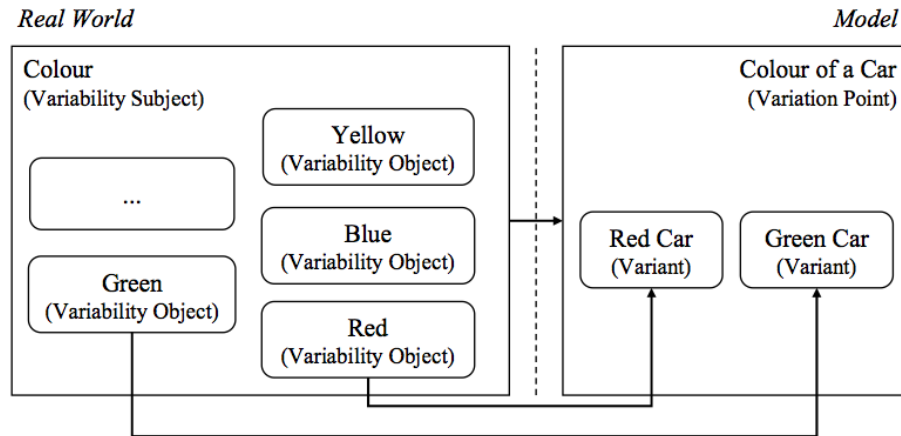


Figure 3.5: Relation between variability in real world and in a model of the real world [PBvdL05].

the possible variants would be represented as specific techniques to implement that persistence type, like SQLite database, in-memory cache, XML, JSON, flat files and so on.

### 3.2.2 Dimensions and Types of Variability

#### Variability in Time versus Variability in Space

In [PBvdL05] two different dimensions of variability can be found. The space dimension is describing existence of assets with different shapes among products. Therefore, variability in space describes variability as introduced before.

The existence of different versions of an artifact at different times, is called variability in time [PBvdL05][BFG<sup>+</sup>01]. This variability dimension is handling evolution of artifacts over time, caused by improvements and error treatments. Configuration management is used to record which version of the artifact is used in which particular product. It may be, that during a design phase a certain functionality may not occur as a different variant, but may be in the future. Therefore this functionality should be designed with a variation point, although it would not be necessary at this moment. As an example, [PBvdL05] mentions a home automation system. This system has a function to identify the user by a magnetic card. Since the designers of this home automation system expected a technological progress in the area of identification mechanisms they defined a variation point for the door lock identification mechanism. Therefore, they can easily handle evolution in this area, for example if there may be the variant of a fingerprint scanner identification mechanism in the future.

While the variability in time specifies the change of an artifact over time, variability in space covers the use of assets with different shapes at the same time. Nevertheless, both variability dimensions occur simultaneously in most of the software product line projects.



### 3.2.3 Internal vs. External Variability

Another way of categorizing variability is introduced in [PBvdL05], this categorization focuses on the visibility of variability. External variability contains variation points that are exposed to the customer, whereas internal variability is containing variation points that are visible for internal usage. Following use cases for internal and external variability exist [PBvdL05]:

- **Internal Variability:**
  - Complexity hiding: It is not necessary to expose all of the complexity of a system to the customer. The customer may be overwhelmed by it and probably associating this complexity with a system that is hard to use.
  - Information hiding: Sometimes it may be useful to hide some functionality, technical details and innovative ideas from the competitors.
  - Fine-grained control: Some external variation points may have more fine-grained options behind and external choice. For example, a person is selecting one painting that should be placed on his wall in his flat (the variation point would be *Painting on Wall*, the variants would be represented as different paintings). The person only wants to decide which painting should be placed on the wall. He might not be interested in picking the right screw for the size of the painting and the kind of the wall. The choice of screw in this example would represent a internal variability.
- **External Variability:** External variability shows the strengths of a software product line and explains which variants of the software product line can be assembled. Therefore it is important to define some variability as external to point out the ability of the system. External variability may also have an effect on marketing. The external variability gives the customer of a software product line an overview of the possible features of one product of this software product line.

### 3.2.4 Variability Mechanisms for Components

When it comes to developing assets the developer has to decide how to implement the variability of a component. There are several methods provided and analyzed in the literature, every single method with different characteristics. The variability mechanisms also differ regarding the time where a variation of a variation point gets introduced.

In [JGJ97] and [GA01] some variability mechanisms are described. [CN02] summed them up and extended the list of variability mechanisms. They are categorized regarding their time of specialization as described below:

- **Requirements Time:**
  - **Extension:** This mechanism is used to reuse an existing use case of a system/component by adding it to another use case and gets applied at requirement time.
  - **Uses:** Another mechanism is represented by the uses-mechanism. In this case, one use of a system is included into another one, therefore one component uses another component to achieve its goal.

- **Implementation Time:**

- **Inheritance:** The component gets specialized by class inheritance, therefore methods can be overwritten, added or extended. This specialization has to be done at class definition time, however, more sophisticated methods can be applied to get a more dynamic approach to that method.
- **Design Patterns:** Design Patterns help to structure code and parts of an application, for example to provide techniques for separation of concerns. [Sch95] describes how to use design patterns to develop reusable software, as well as [GHJ<sup>+</sup>95], representing one of the most cited works regarding design patterns and their impact on reusability.
- **Parameters:** Parameters can be used to configure the behavior of an asset to establish variability. During development it has to be figured out, which properties of an object should be exposed or defined to use with this variability mechanism.
- **Template Instantiation:** Another elegant way to introduce variability is the template instantiation, which allows the specification of unbound elements. These elements get bound when the template gets used. This has to be considered at implementation time, the binding will happen at runtime.
- **Delegation:** Delegation is used as an object-oriented technique to forward requests to other objects (the so-called delegate) which can not be handled by the object itself. Therefore the object holds a reference to the delegation objects and forwards the corresponding calls to the delegate. The operations that can be forwarded are often defined in a protocol and can be marked as optional or required, depending on the used programming language. To implement variability with this delegation pattern, standard functionality is implemented in the delegating property, the variant functionality in the delegate.
- **Object Properties:** A special kind of delegation can be achieved, when certain functionalities are forwarded to a property of an object which claims to implement a specific interface.
- **Frame Technology:** Paul G. Basset invented this technology to manage application in so-called frames [Bas96]. Following this approach, a parent frame is used to copy and/or adapt children frames, whereas these frames can be represented as source files including preprocessor-like directives. When used later in a software product line development life cycle, the restructuring of the code can be very difficult. Therefore, the usage of this method should be evaluated at the beginning of the life cycle. Various implementations of this frame technology exist, for example Netron Fusion<sup>1</sup>, which is specialized in the area of business software, or XVCL<sup>2</sup>, an XML-based open-source implementation of frame technology.

---

<sup>1</sup><http://www.netron.com>

<sup>2</sup><http://xvcl.comp.nus.edu.sg>

- **Runtime or Compile Time:**

- **Configuration:** There is also a way to configure assets at runtime with a separate resource file, like the JavaBeans property file. The component has to know how to handle this configuration file and how to provide functionality to consume this file.
- **Generation:** A very common method in scientific researches to introduce variability to software product lines is the method of generation. With intelligent tool support the specific fragments get generated at compile time or even during runtime.
- **Static Libraries:** Static libraries are linked to an application after compilation and get loaded into the same memory. They contain a set of external functions, which can be used in the application because of the fact that the signatures of these functions are known to the compiled application code. The application can select different libraries and therefore variability can be achieved to a certain degree.
- **Dynamic Linked Libraries:** In contrast to static libraries, dynamic linked libraries (DLL's) are loaded at runtime when needed, hence a more sophisticated way of variability can be provided compared to static libraries.
- **Dynamic Class Loading:** This technique is heavily used in Java, where classes are loaded into memory at the time they are needed inside the program. This feature can be used to decide which class to load at runtime.
- **Conditional Compilation:** Another way to achieve variability is established by the use of preprocessor directives which are handled at pre-compile time. These directives mark the variation point in the code and are easy to use in ordinary use cases, however, complex situations like recursions are hard to manage with this technique.
- **Overloading:** By overloading existing names are used, but the functionality gets modified to allow working with different types. This technique can be applied to functions, procedures or operators and occurs at compile time.
- **Reflection:** Reflection is a technique to modify the structure and behavior of objects during runtime, like values, properties and functions. Therefore, meta information is saved in binary code, to get knowledge about classes, instances or methods.
- **Aspect-oriented Programming:** Aspect-oriented programming [KLM<sup>+</sup>97] is a programming paradigm to increase the extensibility and modularity of a program by specifying cross-cutting concerns (e.g. logging). In Aspect-oriented programming pointcuts are specific points in the execution plan of a program, where additional code (the so-called advice) can be executed. This combination of advice and pointcut is called aspect.

### 3.3 Variability Modeling Techniques

This chapter describes different of modeling techniques to express variability of a software product line. Furthermore, one technique will be chosen to build a variability model (in Chapter 6) of the examined requirements (see Chapter 2).

The described modeling approaches were found in [IKPJ11], which separates the approaches into two groups. The first group of models are combining the representation of the assets and the variability into one model. The second group is working with different models for assets and variability representation.

#### 3.3.1 Feature Models

Feature models were first introduced in the *Feature-Oriented Domain Analysis* in the year 1990 and are probably the most used modeling approach for variability models. There also exist a number of extensions to meet special needs or enhance expressiveness. In FODA (see [KCH<sup>+</sup>90]) a feature is defined as “*prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system*” and is part of a feature model, represented as feature-tree in a diagram. Such a feature diagram in tree notation is illustrated in Figure 3.6, as well as an explanation of the used notations.

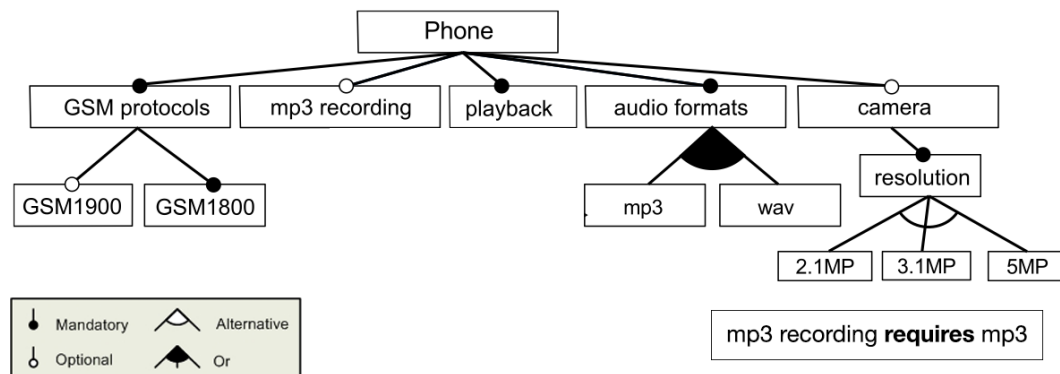


Figure 3.6: A feature model to represent a phone system in tree notation [CGR<sup>+</sup>12].

A feature diagram starts with a root feature, representing the system domain, and can have several child-nodes, also represented as features. These can be mandatory or optional. A relationship from a parent feature to its child can be marked as alternative (xor – exactly one feature has to be selected), or can be marked as or-relationship (at least one of the features has to be selected). Additionally, there are textual additions to express cross-tree constraints, like require- or exclusion-clauses.

Below, the two well-known extensions of *FODA* feature models, *Cardinality-Based Feature Modeling* and *Orthogonal Variability Model* are discussed, to highlight their characteristics.

### Cardinality-Based Feature Modeling

The cardinality-based feature modeling combines several extensions of the original *Feature-Oriented Domain Analysis* notations [cza]. An example is shown in Figure 3.7. The notation is shown in Table 3.1, the used extensions in these systems are described in [CK05] and listed below.

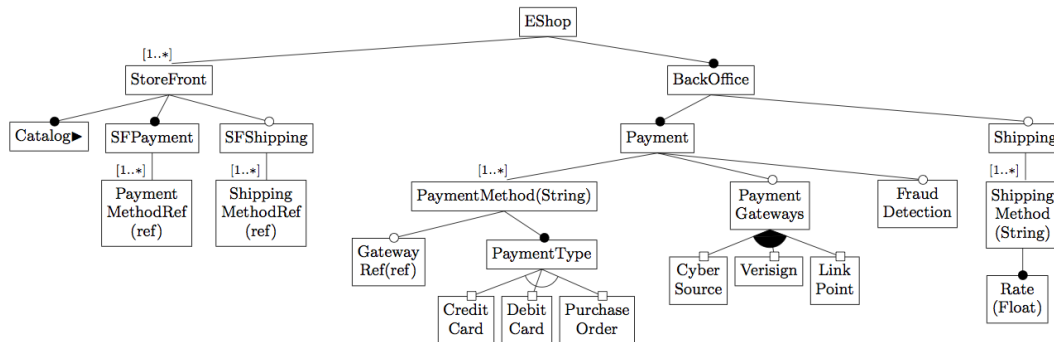


Figure 3.7: Sample of a cardinality-based feature model [CK05].

- Feature Cardinality:** Every feature of a cardinality-based feature model has a parameter which indicates the cardinality of the feature. This cardinality is represented as an interval where the upper bound has to be greater than or equal to the lower bound of the interval, or can be represented as an unbound border with the Kleene star \*. Therefore, it is possible to clone features for a certain configuration. In the example shown in Figure 3.7 the feature *PaymentMethod* is represented as such a feature with unbound upper border, thus a configuration of an *EShop* can have multiple *PaymentMethods*, e.g. one for credit cards and one for debit cards. Solitary feature with cardinality [1..1] and [0..1] can also be expressed by an filled or unfilled circle. In this case, the interval must not to be explicitly mentioned in the model (see Table 3.1).
- Feature Groups:** Cardinality-based feature modeling provides a method to group features into feature groups with a group cardinality. This group cardinality is an interval of the form  $\langle m-n \rangle$ . The value  $m$  is the lower bound and has to be greater than or equal to zero and  $n$  represents the upper bound and has to be greater than or equal to the lower bound. This upper bound has also to be lower than or equal to the total number of features within the group. Therefore the group cardinality specifies how many features of the group can or must be selected. In the example given in Figure 3.7, *Payment Gateways* represents a feature group, where at least one feature has to be selected, whereas *PaymentType* represents a feature group where exactly one feature has to be selected.
- Attribute Types for Features:** Another useful extension is the ability to have attribute types for features, which allows to specify their value during configuration. The type of such an attribute can be a basic one, such as *String* or *Integer*, or a reference to another







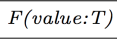




Symbol	Explanation
	Solitary feature with cardinality $[1..1]$ , i.e., <i>mandatory</i> feature
	Solitary feature with cardinality $[0..1]$ , i.e., <i>optional</i> feature
	Solitary feature with cardinality $[0..m]$ , $m > 1$ , i.e., <i>optional clonable</i> feature
	Solitary feature with cardinality $[n..m]$ , $n > 0 \wedge m > 1$ , i.e., <i>mandatory clonable</i> feature
	Grouped feature with cardinality $[0..1]$
	Grouped feature with cardinality $[1..1]$
	Feature $F$ with attribute of type $T$ and value of $value$
	Feature model reference $F$
	Feature group with cardinality $\langle 1-1 \rangle$ , i.e., <i>xor-group</i>
	Feature group with cardinality $\langle 1-k \rangle$ , where $k$ is the group size, i.e., <i>or-group</i>
	Feature group with cardinality $\langle i-j \rangle$

Table 3.1: Symbols used in cardinality-based feature modeling [CK05].

feature, a so-called *feature reference attribute (FRef)*. In the example shown in Figure 3.7 the feature *PaymentMethod* has an attribute of type *String* to specify the names of the different payment methods. An example of a *feature reference attribute* is represented by the feature set *SFPayment*, which points to another feature, in that case it should be a payment method feature. Since this feature reference can point to every other feature, constraints have to be set up to establish a consistent behavior.

- **Feature Model References:** Feature models can become confusing when they are reaching a certain amount of features. Feature model references allow to reference to another feature model, therefore parts of the model can be split up to enhance readability. Since a model can have more than one reference to the same feature model, subscripts are used when it comes to unfold and copy referenced feature models. In the example in Figure 3.7, *Catalog* refers to another feature model.

- **Constraints:** Simple constraints, like the *implies* or *exclude* constraint were introduced very fast as an addition to the tree-based structure of feature models. Through the introduction of cardinalities, attributes and cloning, a more sophisticated way to express constraints had to be found. One approach is to add an additional context to the well-known *implies* and *exclude* constraints. A more flexible and powerful way to define constraints was also determined, therefore the *Object Constraint Language* (OCL)[Obj13] got analyzed whether it meets the needs of expressiveness of such constraints in a feature model in [CK05], as well as XML Path Language (XPath[Wor13]) in [AC04]. An example of an OCL-based constraint set is given in Listing 3.1, related to the example from Figure 3.7. The first constraint is checking if a used *PaymentMethodRef* is indeed in the list of provided payment methods. Hence this constraints is handling the type check of the feature reference. The second one describes an upper bound check for a cardinality. Therefore the example was slightly changed in so far as the constraint restricts the total number of payment methods of the store front to three, instead of allowing unlimited payment methods.

Listing 3.1: OCL-based constraints [CK05].

```

1 context PaymentMethodRef inv:
2   EShop.BackOffice.Payment.PaymentMethod->includes(att)
3
4 context StoreFront inv:
5   SFPayment.PaymentMethodRef->size() <= 3

```

### Orthogonal Variability Model

Another approach to model variability in a software product line is the orthogonal variability model. In [MHP<sup>+</sup>07] and [SvGB05] two kinds of variability are discovered, which are not covered by the mentioned classifications in Chapter 3.2. These two kinds are separating variability into the following parts:

- **Software Variability:** “Ability of a Software system or artifact to be efficiently extended, changed, customized or configured for use in a particular context”[SvGB05]. Hence software variability in the sense of the given citation is well known from the development of single system and is covering object-oriented programming topics, like the usage of abstract superclasses and its specializations as subclasses, or the usage of interfaces and their different implementations.
- **Product Line Variability:** This part of variability in a software product line is specifying the variation between systems, like features or requirements that are fulfilled.

This separation tries to split up the variability into a lower level variability (i.e. source code related), which is not that important for orthogonal variability models, and a higher level variability (i.e. feature or requirement related), which describes the variants of a system. To separate

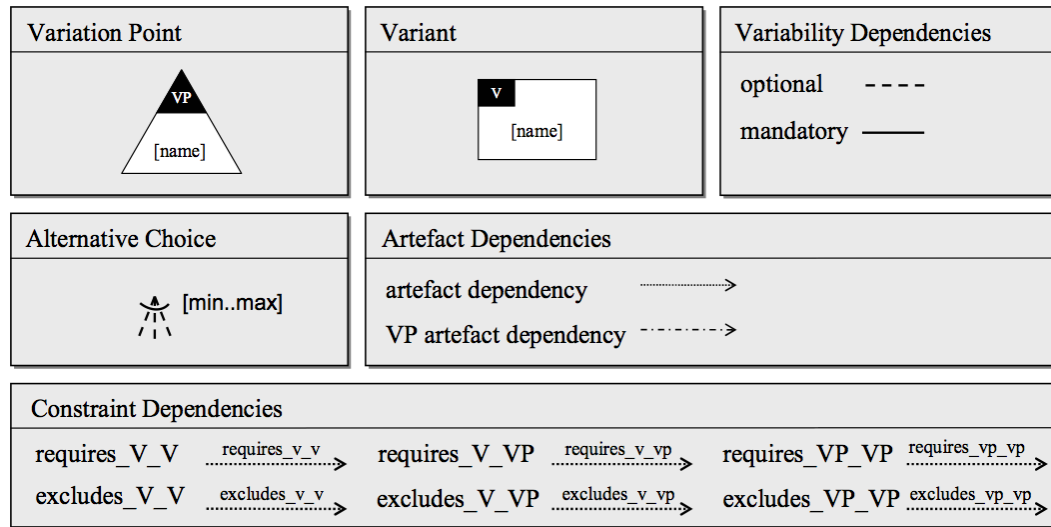


Figure 3.8: Graphical notation for orthogonal variability models [PBvdL05].

these two kinds of variability, the usage of orthogonal variability models to describe product line variability and feature models to document software variability are proposed in [MHP<sup>+</sup>07]. The mapping of product line variability to software variability is discovered as a challenging part in a software product line development process in [MHP<sup>+</sup>07].

Moreover, another problem is explored regarding the different base models, i.e. component diagrams, test models, use case diagrams, etc. Variability is immanent in all of these models and can be documented by extensions to the specific model languages. Another way to link variability to these artifacts can be achieved by *implemented by* links in FORM (feature-oriented reuse method, [KKL<sup>+</sup>98]) feature models, which connect the product line variability to the implementation technique (software variability). Others are using similar approaches like traceability links or inclusion rules (see [CP06] and [DGR07]).

To summarize, orthogonal variability models can be described as variability models, which relate the variability to base models, such as use case models, design models, component models, test models etc. [PBvdL05]. Therefore they are used to represent variability among different artifacts.

The notion of orthogonal variability models is given in Figure 3.8 and is describing concepts like optional and mandatory variability dependencies, as well as alternative choices and constraints (*requires* and *excludes* constraints). In orthogonal variability modeling constraints can involve variants and variation points, therefore an existence of a variant can cause the existence of a certain variation point and vice versa. To track traceability between the variability model and the corresponding base models, artifact dependencies are used. An example how to represent such a variability model and the traces to a use case diagram is shown in Figure 3.9.

The artifact including the use case diagram is describing the use case *opening the front door*. This may be done by unlocking the door with a keypad, or by fingerprint. The artifact



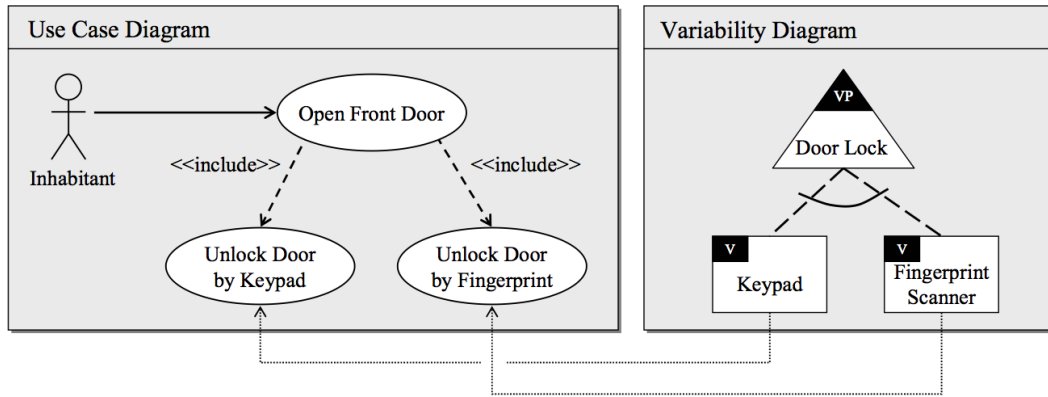


Figure 3.9: Example of orthogonal variability modelling [PBvdL05].

dependency is linking the concrete variant *Keypad* of the variation point *Door Lock* to the use case *Unlock Door by Keypad* and the concrete variant *Fingerprint Scanner* to the use case *Unlock Door by Fingerprint*. This linking may be useful to understand complex variability diagrams, but also adds complexity by adding variability in domain artifacts.

### 3.3.2 Decision Models

Decision models can also be used as variability models and follow a decision-oriented approach to guide the process of product derivation. In [BFG00] decision models are defined as follows:

*A decision model captures variability in a product line in terms of open decisions and possible resolutions. In a decision model instance, all decisions are resolved. As variabilities in generic workproducts refer to these decisions, a decision model instance defines a specific instance of each generic workproduct and thus specifies a particular product line member. [BFG00]*

In [CGR<sup>+</sup>12] the *Synthesis method* [Cor93] developed from the Software Productivity Consortium is mentioned as the first approach of decision modeling and is placed as ground work for all other, later developed decision-based approaches. This method defines decision models as follows:

*Set of decisions that are adequate to distinguish among the members of an application engineering product family and to guide adaptation of application engineering work products. [CGR<sup>+</sup>12]*

This definition shows that decision models focus on the product derivation, whereas feature models (see 3.3.1) are describing the domain [CGR<sup>+</sup>12].

While comparing the feature model in Figure 3.6 and the decision model from the same system (e.g. Table 3.2 or Listing 3.2), one can clearly see that feature models also express the

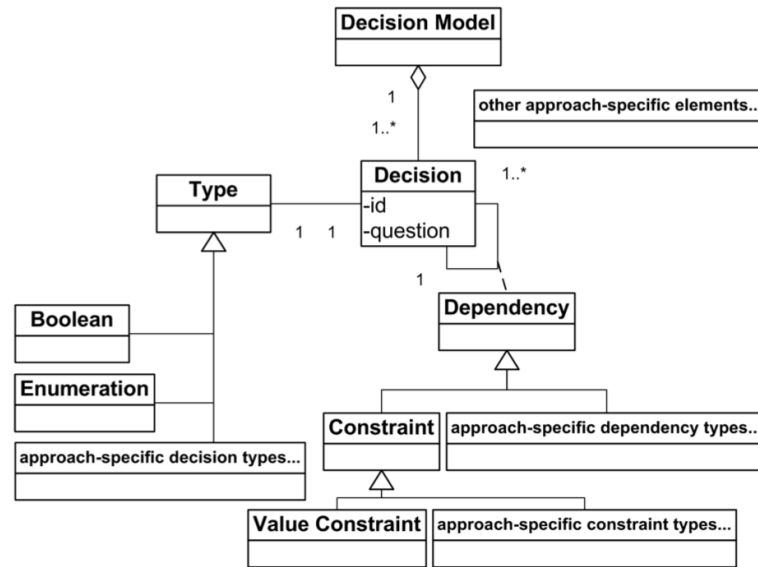


Figure 3.10: UML class diagram of common decision model elements [SRG11].

commonalities of a system, whereas decision models are hiding this aspect [CGR<sup>+</sup>12]. In the example in Figure 3.6 the mandatory feature *playback* is such a hidden feature.

In [SRG11] different decision model approaches were analyzed. They tried to derive a generic structure of a decision model by comparing the different elements of the examined decision models. After this comparison they set up a UML class diagram, which represents the common parts of a decision model. This diagram is shown in Figure 3.10. The examined approaches for this generic UML class diagram are the above mentioned *Synthesis*, an approach from *Schmid & John*, *DOPLER*, *VManage* and *KobRA*. All of the discovered approaches build their models around a set of *decisions*, which have some attributes, varying through the different modeling approaches, but at least have an identifier and a question, representing a manner understandable by an end-user. The *type* of answer given to that question can be of a certain type, most approaches allowing at least an enumeration of answers, but also boolean values for yes/no-answers or specific answer types.

The thing that differs most, is the way how the different approaches handle *dependencies* among decisions [SRG11]. At least some kind of value constraints are given in every of them, but more sophisticated dependency models are common as well, like constraint handling, boolean expressions or other condition- or rule-based systems.

There are several notations for decision models, a tabular notation is shown in Table 3.2, a textual notation for *Synthesis* is represented in Listing 3.2 [CGR<sup>+</sup>12].

Decision Name	Description	Type	Range	Cardinality Constraint Relevant if
GSM_Protocol_1900	Support of GSM 1900 protocol?	Boolean	true false	- - -
Audio_Formats	Which audio formats shall be supported?	Enum	WAV MP3	1:2 - -
Camera	Support for taking photos?	Boolean	true false	- - -
Camera_Resolution	Which camera resolution is required?	Enum	2.1MP  3.1MP  5MP	1:1 - Camera == true
MP3_Recording	Support for recording MP3 audio?	Boolean	true false	- Audio_Formats.MP3 = true -

Table 3.2: Decision model of a phone system in a tabular notation [CGR<sup>+</sup>12].**Listing 3.2: Decision model of a phone system in the textual notation of Synthesis [CGR<sup>+</sup>12].**

```

1 GSM_Protocol_1900: one of (GSM_1900, NO_GSM_1900)
2 Audio_Formats: list of (WAV, MP3)
3 Camera: composed of
4   Presence: one of (Camera, NO_Camera)
5   Resolution: one of (2.1MP, 3.1MP, 5MP)
6 MP3_Recording: one of (MP3, NO_MP3)
7
8 Constraints
9   Resolution is available only if Presence has the value Camera
10  MP3_Recording requires that also MP3 Audio is supported

```

**3.3.3 Modeling using UML Extensions**

[Cla01b], [CJ01], as well as [KL07] analyzed the possibility to use UML (Unified Modeling Language) to express the variability of a software product line. UML is designed to model a single software system, but also provides methods to extend the UML standard. The above mentioned papers are using UML profiles to extend the UML standard. With this profiles it is possible to create generic models, which are used in domain engineering to describe the architecture of a product line and contain modeled variability. For each product of the product line an instance of this generic model is created.

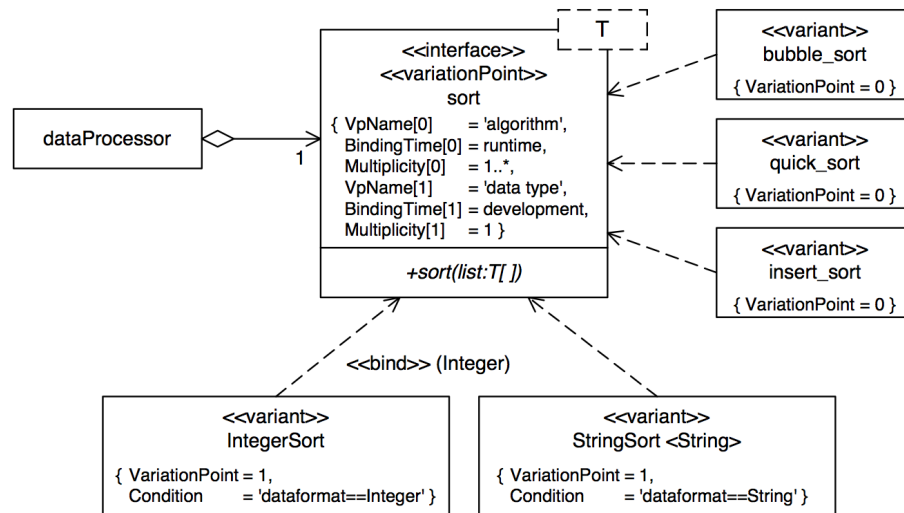


Figure 3.11: Example for modeling with UML extensions including multiple variation points [Cla01c].

Variation points locate a variability and describe several variants, each of them representing one way to realize that variability. For the UML extension the three parts of the variation point are used to model the variation point. These three parts are the location of the variability (the name of the variation point), the different variants of the variation point and a relationship between every variant and the variation point, which assigns every variant to exact one variation point. To distinguish between these parts, stereotypes are used, namely the stereotype `«variationPoint»` and `«variant»`. Model elements which contain variability are applied to the stereotype `«variationPoint»` instead of introducing another virtual model element and can therefore be directly used for code generation.

Both stereotypes can be applied on classes, components, packages, collaborations and associations. `«variationPoint»` also implies some tagged values to specify different binding times and multiplicity of variants. To express interactions between variants and other parts of the model, dependencies are introduced e.g. for mutual exclusions (`“mutex”`) and `“requires”`-statements and also evolution is kept in mind by the stereotype `«evolution»`. If a component changes, the `«evolution»` stereotype describes the interaction between the different versions of the component. Therefore, the stereotype `«evolution»` defines whether the relation between different versions of a core assets has a replacing, extending or decomposing character [Cla01a].

If it comes to multiple variation points, every variation point needs an unique name by specifying a tagged value in the variant that contains the index of the variation point.

The example shown in Figure 3.11 represents two variation points, one for the algorithm used for sorting the data and another one to specify the data type of the elements to sort, the last one including a condition-based on the format of the data. Both of them are specifying different binding times, which can be set to *development* time, system *build*, *installation* time, system *startup* or *runtime*. In this example, the implementation technique for the first variation

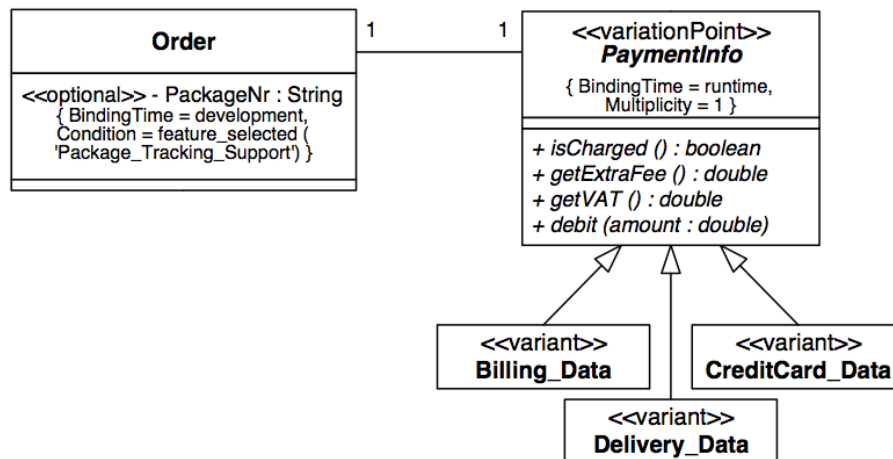


Figure 3.12: Example for modeling with UML extensions using generalization technique and optional attribute [Cla01c].

point is not specified and therefore the variations are connected by dependencies to the variation point. The second variation point is realized using parameterization and thus the variation point is modeled as a template class and the variants are assigned by template bindings.

In Figure 3.12 an example is given how to use the implementation technique generalization. This technique is used to express the different payment methods, as well as an optional variant to specify a tracking number, if a specific feature is selected. To check if this condition evaluates to true, the function `feature_selected()` is used, which checks the existence of a specific feature, in that example the existence of the feature `Package_Tracking_Support`. It is also possible to mark classes or even packages as optional, which leads to a removal of unused parts of the system during compile time, if the right binding time is selected.

### 3.4 Software Product Lines and Variability Mechanisms in Mobile Context

The before examined research gives an overview in general software product engineering topics. This section describes some research with context in the mobile area and how variability is used in this context.

[Jaa02] was written in 2002 and describes the development and thoughts behind mobile browsers in the product line of Nokia phones. [Jaa02] describes the risks that come with a product line, as well as the benefits which may be gained from its usage. The focus is set on the development of an application (in this case a mobile browser) for a software product line. Therefore, the phone itself is the product line, whereas the mobile browser is just a feature, which has to be adopted for each product of the software product line, due to their technical limitations and capabilities.

In contrast, this thesis is focusing on an application as product line, whereas the phone or tablet is just the environment where this product line is running on. It is not focusing on porting this product line from a tablet to a phone, from iOS to Android or web to desktop, but on the variants of the product line as a software product.

How to build a software product line for a mobile device is covered by [You05]. The goal of this thesis was to show how a software product line can be built using *J2ME* and *AspectJ*. The application is represented as a photo browser with mandatory features like creating a photo album, view, add, delete and label a photo and optional features like sending a photo via SMS or email, link photos to address book contacts and so on. The features of the variants were defined by analyzing the device's capabilities and restrictions. The software product line was built with the aspect-oriented programming framework *AspectJ* and another time with object-oriented programming patterns. The results were compared regarding size of the application, heap memory allocations, source code metrics, scalability, modularity, flexibility and complexity of the implementation. They figured out that an aspect-oriented approach, as supported by *AspectJ*, leads to improvements regarding complexity, when it comes to a lot of optional features. As a negative side-effect the increase of the application's size is mentioned, which was a big concern back in 1999, when [You05] was written.

Due to the fact, that mobile devices do not have to struggle with application size nowadays, and the absence of aspect-oriented approaches (see below) for Objective-C, the results of the thesis [You05] can not be applied for this thesis.

*J2ME* and *AspectJ* were also used in [AMJC<sup>+</sup>05]. They extracted a product line from different games for mobile devices and built the variants using an aspect-oriented approach. This work is focusing more on the extraction process and is creating a product line derived from given products, instead of deriving products from a product line.

When searching for aspect-oriented programming approaches for *Objective-C*, several libraries can be found:

- **AspectCocoa**<sup>3</sup>: 1 commit, 6 stars, last commit December 2011
- **FlexOC**<sup>4</sup>: 22 commits, 2 stars, last commit May 2012
- **AOP-for-Objective-C**<sup>5</sup>: 10 commits, 14 stars, last commit May 2012
- **AspectObjectiveC**<sup>6</sup>: 54 commits, 27 stars, last commit May 2011

The number of GitHub stars is an indicator for the popularity and the level of awareness of repositories. All above mentioned libraries had their last commit over one and half year ago. These two facts let assume that no more work will be put in these libraries. If a documentation is provided, it generally says that the libraries were built as an experiment only and therefore a lot of known bugs are still in these libraries.

Additionally, an integration of these libraries seems complex and inscrutable. Therefore the usage of these projects for this thesis was omitted.

However, the found research papers underline the fact that most papers which are covering concrete implementations of software product lines in mobile context are using *Java*-based technologies and aspect-oriented approaches. All these papers are covering older topics like J2ME, newer ones which are describing software product lines for smart phones, tablets or multi-touch devices in general could not be found. In addition, scientific papers analyzing implementation mechanisms for software product lines in *Objective-C* could not be found either.

---

<sup>3</sup><https://github.com/bracken-dev/AspectCocoa>

<sup>4</sup><https://github.com/pvantrepote/FlexOC>

<sup>5</sup><https://github.com/ndcube/AOP-for-Objective-C>

<sup>6</sup><https://github.com/tomdalling/AspectObjectiveC>





# Objective-C Principles

In this Chapter, principles of Objective-C are introduced to create a basic knowledge about the programming language. This knowledge base will be used later in Chapter 5 to show how to implement variability mechanisms with Objective-C and to develop tools that use these variability mechanisms in combination with object-oriented programming techniques, design patterns and the ability of Objective-C.

## 4.1 Key-Value Coding (KVC)

Key-value coding describes a mechanism allowing applications to access properties of objects indirectly by name (the so-called key), rather than directly through invocation of their accessor methods or as instance variables [App13c].

The properties are accessed by using strings to identify the name of the property. Properties with object values, scalar types and structures are supported by key-value coding. To gain the support of scalar types and structs, the parameters and return types are detected automatically and wrapped or unwrapped to an object value. A special type of a key is a key path. A key path is a string of dot separated keys to specify a sequence of object properties.

An example of such a key path, as well as a general example of the usage of key-value coding is given in Listing 4.1. In this example, an instance of the class **MyClass** holds another instance of the same class in the property `linkedInstance`. On line 22 a value is set for the property `integerProperty` of the property `linkedInstance` by using the key path `linkedInstance.integerProperty`.

Key-value coding can often be used to simplify code. For example it is possible to generalize implementations, provide additional code to validate values for specific properties, handle special behavior when setting nil values, or to use built-in operators for collections inside a key path to calculate the maximum, minimum, etc. of the collection. In Listing 4.2, the method `-(BOOL)validateName:error:` gets called automatically when setting the name of the object, which contains this method. In that case, the value will get validated and an error will occur

if this value is `nil` or the length of the string value is smaller than two. The lines 11 to 15 are showing an example usage of operators with key-value coding. `transactions` is an `NSArray` of transaction objects, which hold a parameter called `amount` and a parameter representing the payee of the transaction. On line 11, the average of all amounts of the transaction is calculated, whereas line 12 is counting all transaction objects. Line 15 is retrieving all distinct payees of all transactions.

Key-value coding is efficient, but adds a level of indirection and is slightly slower than direct method invocations. Therefore it should only be used when a benefit from key-value codings flexibility can be achieved. Another thing to keep in mind is the fact that the method `-(id)valueForKey:` is caching Objective-C runtime information and therefore should not be overridden, otherwise these advantages gained from caching will be lost.

**Listing 4.1:** Example to illustrate access through accessors and access through key-value coding [App13c].

```

1 @interface MyClass
2
3 @property NSString *stringProperty;
4 @property NSInteger integerProperty;
5 @property MyClass *linkedInstance;
6
7 @end
8
9 // Somewhere else ...
10 MyClass *myInstance = [[MyClass alloc] init];
11 myInstance.linkedInstance = anotherInstance;
12
13 // ... using Accessors
14 NSString *string = myInstance.stringProperty;
15 myInstance.integerProperty = 2;
16 myInstance.linkedInstance.integerProperty = 5;
17
18 // ... using KVC
19 MyClass *myInstance = [[MyClass alloc] init];
20 NSString *string = [myInstance valueForKey:@"stringProperty"];
21 [myInstance setValue:@2 forKey:@"integerProperty"];
22 [myInstance setValue:@5 forKeyPath:@"linkedInstance.integerProperty"];

```

---

Listing 4.2: Examples to simplify code with key-value coding [App13c].

```

1 // Providing validation method for name property. The name must not be nil
  , and must be at least two characters long.
2 - (BOOL)validateName:(id)ioValue error:(NSError **)outError {
3     if (*ioValue == nil || [(NSString *)*ioValue length] < 2) {
4         // set outError here
5         return NO;
6     }
7     return YES;
8 }
9
10 // Simple collection operators
11 NSNumber *transactionAverage = [transactions valueForKeyPath:@"@avg.amount
    "];
12 NSNumber *numberOfTransactions = [transactions valueForKeyPath:@"@count"];
13
14 // Object operators
15 NSArray *payees = [transactions valueForKeyPath:@"@distinctUnionOfObjects.
    payee"];

```

## 4.2 Objective-C Runtime

The Objective-C runtime is a library that acts like an operating system for Objective-C and is built using C and Assembler to add the object-oriented capabilities of the programming language. Objective-C is a runtime-oriented language, which means that decisions regarding binding is shifted to the runtime instead of compile or link time.

Hence it is possible to add methods to classes or exchange implementations of methods. The redirection of method calls to other classes during runtime is also supported by the runtime. Another feature of the Objective-C runtime is to retrieve information about classes or methods an object is implementing. With this technique the application can also determine which classes are conforming to a particular interface [App13e].

One big difference compared to other languages is the fact, that Objective-C is using messaging. Therefore a developer is not calling a method of an object, but is sending a message to this object. If the object does not recognize this message, the Objective-C compiler will flag this line of code with a warning. When executing this code, it will cause a runtime exception. The compiler will allow to run an application with this warning flags, since the Object-C runtime allows to extend classes and objects with methods dynamically at runtime.

The runtime is used finding the corresponding message to execute and follows the following steps<sup>1</sup>:

1. The class cache and class dispatch table of the class and super-classes of the object the message was sent to are searched for the specific method.

<sup>1</sup>Blogpost by Colin Wheeler, 01-20-2010 <http://cocoasamurai.blogspot.co.at/2010/01/understanding-objective-c-runtime.html>

2. The runtime will call `+(BOOL)resolveInstanceMethod:(SEL)selector` of the class of the object the message was sent to in case the search failed. If this method is implemented, it could provide a special implementation for the sent message.
3. If the last step returns with **NO** (so no method has been resolved), the runtime will call `-(id)forwardingTargetSelector:(SEL)selector` to ask the object if another object can be provided to forward the message.
4. If this step also does not lead to a successful method invocation, an unknown message is sent to the object and the application will crash.

Another functionality of the Objective-C runtime is the ability to add associated objects to an object. With this feature it is possible to add additional variables to a class, including the definition of their store behavior. This might be useful if a predefined class should hold an additional property and subclassing is not intended. Another example is shown in Listing 4.3. In this example a cell of a `UITableView` is configured to have an `UIButton` as a accessory view on line 6 (that is the view presented on the left border of the cell). When this button gets triggered the method `-(IBAction)cellButtonPressed:` gets called. The application will not be able to determine the button's enclosing cell, i.e. the index path of the cell where the triggered button is set as accessory view. One solution is to subclass `UIButton` to add a property of type `NSIndexPath` to store the information about the cell inside the class. This might not be necessary in most of the use cases and therefore this solution will generate an overhead of work. Instead the corresponding index path can be attached as associated object during cell creation (see line 7 of Listing 4.3). This associated object is retrieved in the action trigger method on line 13 to get the index path of the cell.

Listing 4.3: Example usage of associated objects.

```

1 void * const kNSIndexPathKey = "NSIndexPathKey";
2
3 - (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath {
4     UITableViewCell *cell = ...
5     UIButton *button = ...
6     cell.accessoryView = button;
7     objc_setAssociatedObject(button, kNSIndexPathKey, indexPath,
        OBJC_ASSOCIATION_ASSIGN);
8     return cell;
9 }
10
11 - (IBAction)cellButtonPressed:(id)sender {
12     UIButton *button = sender;
13     NSIndexPath *indexPath = objc_getAssociatedObject(button,
        kNSIndexPathKey);
14     ...
15 }

```

## 4.3 Class Extension with Categories

Sometimes it would be beneficially, if existing classes could be modified by adding behavior which is needed in a special use case. In such cases, it does not make sense to add this behavior to the original classes, because in other use cases this additional behavior will not be needed or the extension of this class would mess with a clear class design regarding separation of concerns or in some other cases, the original class might not be modifiable (i.e. in closed frameworks). Objective-C provides a technique to add custom methods to existing classes through categories to accomplish this task (see [App13f]).

The syntax for defining a category is given in Listing 4.4. This category extends a given class `PHMPerson` to easily retrieve the full name of a person. The category on `UIView` in Listing 4.5 is defining shorthands for the view's right, left, top and bottom border. For example, this shorthand allows to retrieve the view's right border by calling the property `phm_frameRight` instead of calculating this border by adding the x-coordinate and the width of the view.

It is also possible to declare a private class extension for properties and methods. This is necessary, if the developer wants to hide some properties or methods, since Objective-C is missing access modifiers like `public` and `private`. To achieve the effect of a `private` access modifier, a category is defined in the implementation file of the class, where private methods and properties can be defined. It is also possible to use this private category to implement properties that are `readonly` from outside but modifiable from inside the class. To implement that effect the property is set to `readonly` in the header file and redefined with the scope `readwrite` inside the private class extension in the implementation file.

Apple is mentioning in their guidelines [App13f], that category methods on Apple framework classes should have a three letter prefix followed by an underscore to prevent name clashes with methods, that might be added to their framework in the future. In the given example it is not necessary to prefix the category on `PHMPerson`, since it is not a predefined class. But following

Listing 4.4: Example of an Objective-C category to extend a self written classs [App13f].

```

1 // PHMPerson+PHMPersonNameDisplayAdditions.h
2 @interface PHMPerson (PHMPersonNameDisplayAdditions)
3
4 - (NSString *)lastNameFirstNameString;
5
6 @end
7
8 // PHMPersonPHMPersonNameDisplayAdditions.m
9 @implementation PHMPerson (PHMPersonNameDisplayAdditions)
10
11 - (NSString *)lastNameFirstNameString {
12     return [NSString stringWithFormat:@"%@", %@, self.lastName, self.
13         firstName];
14 }
15 @end

```

Listing 4.5: Example of an Objective-C category to extend a predefined class [App13f].

```

1 // UIView+PHMShorthands.h
2 @interface UIView (PHMShorthands)
3
4 @property (nonatomic, readonly) CGFloat frameLeft;
5 @property (nonatomic, readonly) CGFloat frameRight;
6 @property (nonatomic, readonly) CGFloat frameTop;
7 @property (nonatomic, readonly) CGFloat frameBottom;
8
9 @end
10
11 // UIView+PHMShorthands.m
12 @implementation UIView (PHMShorthands)
13
14 - (CGFloat)frameLeft { return self.frame.origin.x; }
15 - (CGFloat)frameRight { return self.frame.origin.x + self.frame.size.width
    ; }
16 - (CGFloat)frameTop { return self.frame.origin.y; }
17 - (CGFloat)frameBottom { return self.frame.origin.y + self.frame.size.
    height; }
18
19 @end

```

the guideline the categories on the predefined class `UIView` should have a prefix to prevent future name clashes if Apple is also providing getters for the views right, left, top and bottom border.

## 4.4 Notifications

As an additional improvement in decoupling to the well-known *Observer Pattern*, Objective-C supports a mechanism called *Notifications* [App13d]. The `NSNotificationCenter` is used to broadcast notifications to inform observers about events and works like a notification dispatch table. A subject is posting to the notification center with the method `-(void)postNotificationName:object:userInfo:`, specifying the name of the notification, the object which is causing the event and therefore the notification (in most cases the subject itself) and a dictionary representing additional information about the event, like error descriptions or other relevant information.

The observer is registering at the notification center for a specific notification type (by passing the notifications name). Optionally, it is possible to define one specific object which the observer is interested in. The observer will then receive only notifications issued by this object, but not others. To register an observer the method `-(void)addObserver:selector:name:object:` of the class `NSNotificationCenter` has to be called. The method `-(void)removeObserver:name:object:` is used to deregister the observer for a specific notification.

An example of this steps is given in Listing 4.6. In this Listing an Observer is registering itself as a observer at the notification center for the notification

`StateChangedNotification`. This happens on line 5 in the observer's initializer. When the observer gets deallocated, it will be removed as an observer from the notification center (see line 10). The method `-(void)stateChanged:` gets called whenever such a `StateChangedNotification` is posted to the notification center. On line 23 a subject is posting such a notification.

Compared to the *Observer Pattern*, *Notifications* are loosely coupled. This loosely coupling is achieved because the subject itself does not hold an array of observers, but only informs the `NSNotificationCenter` after the occurrence of an event. The notification center is holding the reference of observers instead and is handling the notification sending. The difference regarding the activity flow is shown in Figure 4.1 and Figure 4.2.

**Listing 4.6: Example of registering as Observer for a notification and sending a notification.**

```

1 @implementation PHMObserver
2
3 - (id)init {
4     ...
5     [[NSNotificationCenter defaultCenter] addObserver:self name:
        StateChangedNotification selector:(stateChanged:) object:nil];
6     ...
7 }
8
9 - (void)dealloc {
10    [[NSNotificationCenter defaultCenter] removeObserver:self name:
        StateChangedNotification object:nil];
11 }
12
13 - (void)stateChagned:(NSNotification *)notification {
14     NSLog(@"Current State: %@", [notification.userInfo objectForKey:@"
        state"]);
15 }
16
17 @end
18
19 @implementation PHMSubject
20
21 - (void)setState:(PHMState *)state {
22     _state = state;
23     [[NSNotificationCenter defaultCenter] postNotificationName:
        StateChangedNotification object:self userInfo:@{@"state": state}];
24 }
25
26 @end

```

---

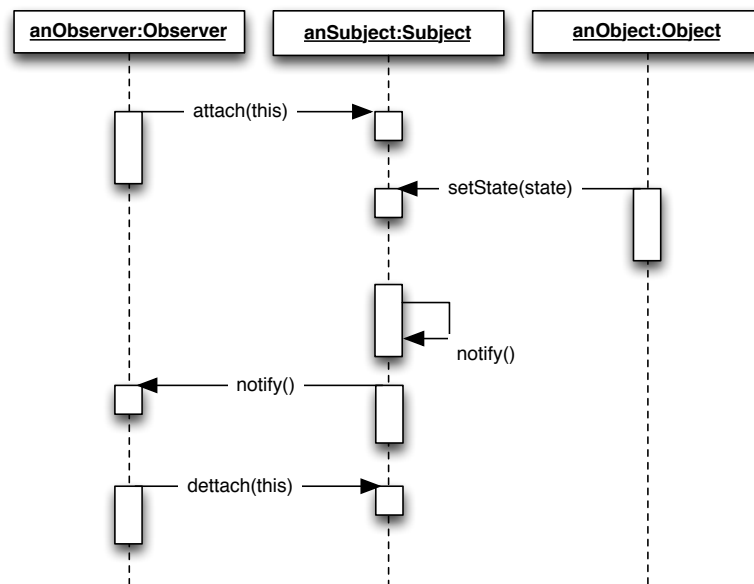


Figure 4.1: Sequence diagram of the observer pattern.

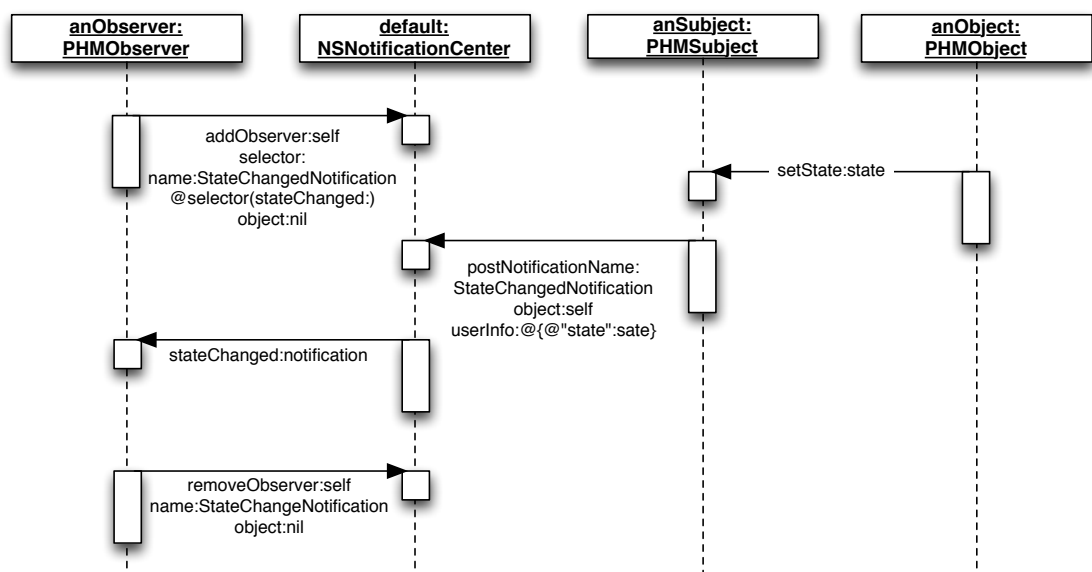


Figure 4.2: Sequence diagram of Objective-C notifications.



## 4.5 UIAppearance

Apple introduced the `UIAppearance` API with iOS 5 to make it easier to configure the appearance of views across an application, without handling all the appearance inside the view implementation itself<sup>2,3</sup>. This makes it easier to change the appearance or to have a consistent appearance across the entire application.

All properties of a view which are annotated with the macro `UI_APPEARANCE_SELECTOR`, can be configured using this API. To determine which properties are marked with that macro the header files of the views have to be investigated. To configure these properties, an `UIAppearance` proxy for the specific view has to be retrieved from the API. After this step, the properties can be set like on an instance of that view. An example of the usage of this API is given in Listing 4.7.

To accomplish a more fine-grained control, it is possible to configure views, which are embedded in a specific containment hierarchy. The Listing 4.7 shows an example, how to set the tint color of all `UIButton`s in the application to green, except for buttons inside the `PHMCustomView`, these are set to yellow.

Without using `UIAppearance`, the configuration of this views will cause boilerplate code to style the buttons. For each button in the whole application the tint color had to be set. One solution is represented by subclassing `UIButton`, but creating a subclass for each style of a button might also cause an overhead and a huge amount of subclasses in the project. Listing 4.7 shows a solution based on `UIAppearance`.

The API also allows to annotate properties of own views with the macro to make this property available for configuration through the appearance API. In the given example the property `myBackgroundColor` of `PHMCustomView` is annotated with the macro and therefore it is possible to set the color with the appearance API to blue.

Sometimes it is annoying to create subclasses for views for the appearance API, just to expose properties that are already existing. Objective-C categories can be used instead to achieve the same goal. In Listing 4.8 the existing property `backgroundColor` of `UIView` gets exposed to the API by defining a new property `phm_backgroundColor`. This new property handles a forwarding to the existing property `backgroundColor`. This forwarding is necessary to be safe regarding API changes when the `backgroundColor` gets annotated with the macro in a future release of iOS.

---

<sup>2</sup>Blogpost by Mattt Thompson, 03-11-2013 <http://nshipster.com/uiappearance/>

<sup>3</sup>Blogpost by Peter Steinberger, 02-12-2013 <http://petersteinberger.com/blog/2013/uiappearance-for-custom-views/>

Listing 4.7: Example usage of the UIAppearance API with custom appearance properties.

```

1 @interface PHMCustomView : UIView
2
3 @property (nonatomic, strong) UIColor *myBackgroundColor
   UI_APPEARANCE_SELECTOR;
4 ...
5 @end
6
7 @implementation AppDelegate
8 ...
9 - (BOOL)application:(UIApplication *)application
   didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
10     ...
11
12     [[UIButton appearance] setTintColor:[UIColor greenColor]];
13     [[UIButton appearanceWhenContainedIn:[PHMCustomView class], nil]
       setTintColor:[UIColor yellowColor]];
14     [[PHMCustomView appearance] setMyBackgroundColor:[UIColor blueColor]];
15 }
16 ...
17 @end

```

---

Listing 4.8: Exposing a given property of UIView to the appearance API with the usage of categories.

```

1 // UIView+PHMAppearanceAdditions.h
2 @interface UIView (PHMAppearanceAdditions)
3
4 @property (nonatomic, strong) UIColor *phm_backgroundColor;
5
6 @end
7
8 // UIView+PHMAppearanceAdditions.m
9 @implementation UIView ()
10
11 - (UIColor *)phm_backgroundColor {
12     return self.backgroundColor;
13 }
14
15 - (void)setPhm_backgroundColor:(UIColor *)backgroundColor {
16     self.backgroundColor = backgroundColor;
17 }
18
19 @end

```

---

# Implementing Variability in Objective-C

This chapter describes how variability mechanisms can be implemented in Objective-C to allow variation among products of a software product line.

The mechanisms described in Section 3.2.4 are used to create methods using object-oriented programming techniques and design patterns with respect to the abilities of Objective-C. The developed methods are acting as tools to create a software product line with Objective-C. They are applicable not only to the collected requirements but are valid for each product line which is implemented with Objective-C.

## 5.1 Inheritance

Subclassing is a variability mechanism that can be easily applied to establish variation among a variation point. Each variant of a variation point is represented as a subclass of this object. A way has to be found to configure which subclass of this object should be used for one specific product. A dynamic approach for this configuration is desirable. Such an approach is developed in this section and described as follows.

### 5.1.1 Instantiating Objects

Most objects are instantiated by calling the class method of `+(id)alloc` of `NSObject` to allocate the needed memory and receive an instance of the needed class, followed by the call of an initializer (e.g. `-(id)initWithNibName:bundle:` from `PHMViewController`). Some classes offer shorthands to get an instance of the class. In the example given in Listing 5.1 it is assumed that `PHMViewController` is providing a method `+(id)viewControllerWithNibName:bundle:` to create an instance, instead of calling `+(id)alloc`, followed by `-(id)initWithNibName:bundle:`.

Listing 5.1: Example of simple subclassing mechanisms with the usage of a parameter.

```

1 PHMViewController *viewController = nil;
2 PHMViewController *anotherViewController = nil;
3
4 // Example 1
5 if (shouldNotUseSubclass) {
6     viewController = [[PHMViewController alloc] initWithNibName:nil bundle:
7         nil];
8     anotherViewController = [PHMViewController viewControllerWithNibName:
9         nil bundle:nil];
10 } else {
11     viewController = [[PHMSpecialViewController alloc] initWithNibName:nil
12         bundle:nil];
13     anotherViewController = [PHMSpecialViewController
14         viewControllerWithNibName:nil bundle:nil];
15 }
16
17 // Example 2
18 Class viewControllerClass = (shouldNotUseSubclass ? [PHMViewController
19     class] : [PHMSpecialViewController class]);
20 viewController = [[viewControllerClass alloc] initWithNibName:nil bundle:
21     nil];
22 anotherViewController = [viewControllerClass viewControllerWithNibName:nil
23     bundle:nil];
24
25 // Example 3
26 Class viewControllerClass = NSStringFromClass(classNameOfClassToBeUsed);
27 viewController = [[viewControllerClass alloc] initWithNibName:nil bundle:
28     nil];
29 anotherViewController = [viewControllerClass viewControllerWithNibName:nil
30     bundle:nil];

```

Example 1 in Listing 5.1 shows an approach, where the selection of the right subclass is handled with a boolean property. If there were more different subclasses involved, an enum structure should be used instead of the boolean property, to determine which subclass should be used in case of subclassing. To instantiate a subclass of a given class, the only thing needed to change in the code is the corresponding class which will receive the message `alloc`. Hence `alloc` will return an instance of the subclass and the initializer message will be sent to this object.

Example 2 in Listing 5.1 uses this observation to eliminate some boilerplate code. It uses the boolean property to create a class object, which is later used to initialize the right object. Capturing more subclasses will lead to a more complex conditional statement, but it is easier to change the initializer later on, if needed so. Hence it is called on a single spot within the code fragment.

### 5.1.2 Creating the Class Object dynamically

To implement a more dynamic subclassing approach, the class to which the *alloc* message is sent, can be retrieved from a string by calling the function `Class NSClassFromString(NSString *string)`. The Objective-C runtime will look into the class names of all loaded classes to find a match, and instantiate a class object out of this class if found.

Example 3 in Listing 5.1 makes use of this function to retrieve the right class from a `NSString` parameter, which makes the code more flexible and adds the ability to support more subclasses.

### 5.1.3 Custom Class Loader

To create a flexible way of using the above mentioned possibilities of subclassing, a custom class loader is needed to easily retrieve the right class and also to register a certain subclass to use instead of another class.

Figure 5.1<sup>1</sup> shows the method declarations of such a class loader. The class loader provides methods to register a subclass for a specific class to override, methods that allow class objects as input, as well as `NSString`s. In the application's source code, all classes that are allowed to subclass with this variability mechanism, should be instantiated retrieving the needed class from the class loader by calling the method `+(Class)loadClassForClass:(Class)class`. To make this process easier, a category on `NSObject` is provided to create a shorthand for this step (see the implementation of the category in Listing 5.2).

When the application starts up, all subclasses should be registered using the methods provided from the class loader. Additionally, it provides a way to initialize subclasses declared in an external configuration file, represented as *plist* in dictionary format. The entries of this dictionary contain the class to override as key and the subclass to use instead of this class as value.

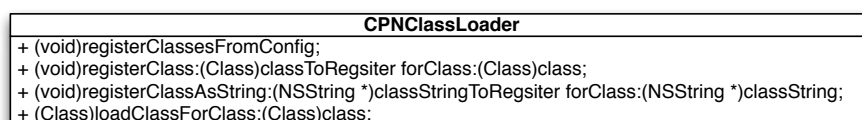


Figure 5.1: Custom class loader to retrieve information about subclassing from a configuration file.

<sup>1</sup>The UML notation was changed slightly for all following class diagrams to improve reading for Objective-C developers. The attributes are using the Objective-C datatypes instead of the defined ones from the UML standard. The visibility of operations was changed to match the notation from Objective-C. + is therefore marking static methods, - is marking public ones. Private methods will not be present in the UML diagram. The notation of operations was adjusted to meet the notation of Objective-C methods, therefore the return type is given at the beginning of the method name, surrounded by brackets. The : character is marking a parameter, the datatype of the parameters is not written down in the diagram to simplify reading.

The class loader itself is represented internally as singleton, which will save the information about the subclasses in a dictionary with the same format as the above described *plist*. When registering a subclass, the class loader will check if the provided subclass is indeed a subclass of the class to override, to prevent conflicts and not resolvable method calls.

The usage of the class loader is explained in Listing 5.3. Through the registration of the class `PHMSpecialViewController` as substitute of `PHMViewController`, instances of `PHMSpecialViewController` are created inside the method `-(void)doStuff`. The only thing to keep in mind using this approach of subclassing as a variability method is that every class which should be replaced by this variability mechanism should be instantiated by retrieving the class method `+(Class)classFromLoader` from the class loader, before instantiating. If needed, additional categories for shorthands to create objects could be very useful, for example a class method called `+(id)viewControllerFromClassLoaderWithNibName:bundle:` in the above given code snippets.

Listing 5.2: Category on `NSObject` and `UIViewController` to provide shorthands to retrieve a class object from the custom class loader.

```

1 @implementation NSObject (CPNClassLoader)
2
3 + (Class)classFromLoader {
4     return [CPNClassLoader loadClassForClass:[self class]];
5 }
6
7 @end
8
9 @implementation UIViewController (CPNClassLoader)
10
11 + (instancetype)viewControllerFromClassLoaderWithNibName:(NSString *)
    nibName bundle:(NSString *)bundle {
12     return [[[self classFromLoader] alloc] initWithNibName:nibName bundle:
        bundle];
13 }
14
15 @end

```

---

Listing 5.3: Custom class loader in practice.

```

1 // in AppDelegate
2 - (BOOL)application:(UIApplication *)application
  didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
3     [CPNClassLoader registerClassesFromConfig];
4     ...
5 }
6
7 //somewhere else
8 - (void)additionalSetupMethod {
9     [CPNClassLoader registerClass:[PHMSpecialViewController class] forClass
      :[PHMViewController class]];
10    ...
11 }
12
13 - (void)doStuff {
14     PHMViewController *viewController = [[PHMViewController
      classFromLoader] alloc] initWithNibName:nil bundle:nil];
15     PHMViewController *anotherViewController = [PHMViewController
      viewControllerFromClassLoaderWithNibName:nil bundle:nil];
16 }

```

Listing 5.4: Setting a bunch of values with a given NSDictionary.

```

1 NSDictionary *personDictionary = @{@"firstname": @"Philip", @"lastname": @"
  Messlehner", age: @(25)};
2
3 PHMPerson *person = [PHMPerson alloc] init];
4 [person setValuesForKeysWithDictionary:personDictionary];

```

## 5.2 Parameters and Configuration

In Chapter 4.1 key-value coding was described and some listings were provided to give some examples of its usage. In these listings the alternative way of setting a property with key-value coding is described through the usage of the method `-(void)setValue:forKey:`. There is another way to batch-set a group of properties, given in form of a `NSDictionary` with the method `-(void)setValuesForKeysWithDictionary:` (see Listing 5.4).

### 5.2.1 Configuration Loader

To combine the above mentioned possibility to batch-set several properties, which are served from a dictionary, the idea of a configuration loader can be derived. Classes, which should be able to be configured, should implement a specific protocol called `CPNConfigurationLoading`. The specification of this protocol is given in Figure 5.2. The class should implement the specific initializer `-(id)initWithConfiguration:` to allow to set up the class with a given configuration

as dictionary. Key-value coding will fire up the method `-(void)setValue:forUndefinedKey:` for each of the provided keys in the dictionary that were not found as property of the object. An assertion will be thrown and the application will crash in case this method is non existent, hence an implementation of this method is also required to handle this case of failure.

Retrieving of configuration dictionaries will be handled from a singleton called `CPNConfigurationLoader`. It provides methods to register a specific class for configuration handling or even an automatic way of registering classes. Therefore the method `+(void)registerClasses` will search for classes which are conforming to the protocol `CPNConfigurationLoading`. This meta information search is done with Objective-C runtime methods. These searched classes have to implement the method `+(NSString *)configurationName`. This name gets used to search for *plist* files in a specific directory of the application and will also be used to save this configuration.

To retrieve the configuration (i.e. for the above mentioned initializer) the method `+(NSDictionary *)configurationForClass:` can be used. The loader will use the value from the method `+(NSString *)configurationName` to search for the right configuration and return it to the caller.

A possibility of saving values (i.e. from a settings area inside the application) is also provided by the configuration loader. Therefore the object, whose values should be persisted, has to implement `+(NSArray *)keyPathsToPersist` and return all keys (or key paths) of properties, which should be saved. This key path will be used to retrieve the value of the object using the key-value coding method `-(id)valueForKey:`. To actively persist a configuration the method `+(void)saveConfigurationForObject:` of the `CPNConfigurationLoader` can be called. Automatic persistence can only be offered to singletons, therefore the loader will search for the initializer `+(instancetype)sharedInstance` for all registered classes to create the singleton and retrieve values from it. An example how to use this concept of configuration loading in real world can be found in Listing 5.5. In this example, the view controller's properties `viewControllerTitle` and `colorValue` will get persisted on line 25 when the view controller disappears. After the view controllers creation, all saved properties will get retrieved from the configuration file in the initializer on line 12. This configuration dictionary gets used on line 17 to set the properties of the class.

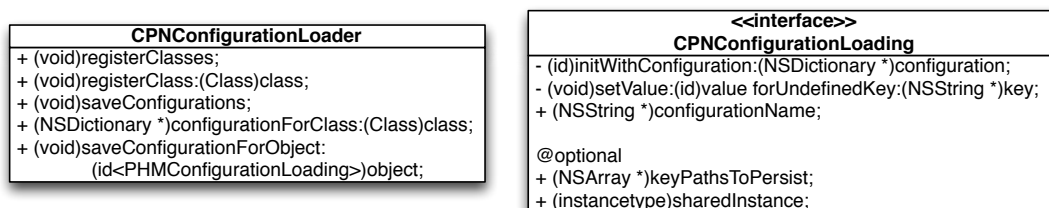


Figure 5.2: Protocol and singleton to implement a configuration loader in Objective-C.



**Listing 5.5:** View controller using configuration loader to set properties on creation and persist them, when disappearing.

```

1 @interface PhMViewController : UIViewController <CPNConfigurationLoading>
2
3 @property (nonatomic, strong) NSString *viewControllerTitle;
4 @property (nonatomic, strong) NSNumber *colorValue;
5 @property (nonatomic, strong) NSString *nonPersistedStringValue;
6 ...
7 @end
8
9 @implementation PhMViewController
10
11 - (id)initWithNibName:(NSString *)nibName bundle:(NSBundle *)nibBundle {
12     return [self initWithConfiguration:[CPNConfigurationLoader
13         configurationForClass[self class]]];
14 }
15
16 - (id)initWithConfiguration:(NSDictionary *)configuration {
17     if ((self = [super initWithNibName:nil bundle:nil])) {
18         [self setValuesForKeysInDictionary:configuration];
19     }
20     return self;
21 }
22
23 - (void)viewWillDisappear:(BOOL)animated {
24     [super viewWillDisappear:animated];
25     [CPNConfigurationLoader saveConfigurationForObject:self];
26 }
27
28 + (NSString *)configurationName {
29     return @"PhMViewController";
30 }
31
32 + (NSArray *)keyPathsToPersist {
33     return @[@"viewControllerTitle", @"colorValue"];
34 }
35 ...
36 @end

```

---

## 5.3 Interceptors

Interceptors are a concept derived from the aspect-oriented paradigm to allow to intercept the applications execution plan and introduce code. These points, where such an interruption can occur, are called events. Some events, that might be useful to use to introduce code, are described below. A definition of the characteristics of interceptors is also provided.

### 5.3.1 Events

The event itself is defined as entering or leaving specific methods. Some examples of such events are described in the following list:

- **Application Start:** Occurs, after the application is started and all initial work is done. It will not delay the application start itself because the so-called watchdog<sup>2</sup> will kill the application and will force a crash. Instead of that, a full screen dialog will show up, that looks like the application-start-image to mime the startup.
- **Application Entering Foreground:** This event occurs, after an application is sent to the background and opened again. To mime the start of the application the same dialog as mentioned before will be used.
- **Synchronization Start:** Occurs, when the synchronization of the data will begin, whether this is caused by a timer or by user interaction (like tapping on a sync button or triggering a pull-to-refresh-action).
- **Synchronization End:** Occurs, after the synchronization of the data ended.

These events should also provide additional information for the interceptors. For example, if the synchronization ended with an error or with a success, how many times the application was opened or entered the foreground, last time of the application being active etc.

### 5.3.2 Interceptor-Types

The introduced code (the so-called interceptor) can have different characteristics and behaviors which require different handling regarding their appearance and execution. The encountered possible characteristics are the following:

- **Concurrency:**
  - **Blocking:** Blocking interceptors cause the application-lifecycle to stop, therefore processes like syncing and other activities should be stopped and continued after execution of the interceptor.
  - **Non-Blocking:** Non-blocking interceptors do not interrupt the application and are allowed to run like background-processes.
- **Visibility:**
  - **Visual (UI):** Visual interceptors represent an user interface element and the ability to interact with it. They could be represented as modal dialog or as embedded user interface element.

---

<sup>2</sup>Apple introduced the watchdog to force developers to move heavy time and performance consuming actions away from the application start to improve the user experience. For example: network request, heavy parsing of large data sets, etc. should be moved to a later point of the application life cycle like the first appearance of a view controller

- **Hidden:** Hidden interceptors are running without any visual feedback in the background and do not provide any input option.

Every interceptor is specified by the combination of his concurrency, visibility and is attached to at least one event.

### 5.3.3 Technical Realization

In Java aspect-oriented approaches are used to implement interceptors. Objective-C does not support such an approach therefore other approaches had to be found to introduce an interceptor mechanism. Three solutions were developed to implement interceptors, but none of these solutions solved the problem satisfiable. The three different approaches are described as follows and are compared to each other to show which approach suits best for the encountered interceptor types.

#### Delegation Forwarding

Objective-C is heavily using the *Delegation Pattern*, to inform other objects about occurrences of events and delegate work. For example each application has a delegate associated, which is executing code after the occurrence of events of the applications life cycle. This delegate has to conform to the protocol **UIApplicationDelegate**, with methods like

– **(BOOL)** applicationDidFinishLaunching: or – **(void)** applicationWillTerminate:. Unfortunately, it is not flexible enough to extend the applications delegate to introduce additional code, that should be executed, as well as a mechanisms for multiple delegates are missing.

As a solution a delegation forwarding can be implemented. The delegate itself therefore holds a reference to one or more additional delegates and forwards the calls to them, after executing its own code. If it is possible to extend the class, which is responsible to call the designated delegate, this forwarding can be implemented in the delegator. The added delegates can decide on their own, whether to execute their code blocking, or in a separate thread (and therefore non-blocking).

As a negative side effect, this forwarding has to be implemented for each of the delegates method, although this method is a very simple one to enable the usage of blocking and non-blocking, hidden interceptors.

#### Notifications

Notifications are used to broadcast information about occurrences of events, hence they could also be used to implement interceptors. Therefore a notification has to be posted on every point in the applications execution plan, which should be intercepted. With this approach it is easy to extend the application with interception points with a decoupled mechanism.

Notifications are commonly consumed asynchronously, but there exists a way to consume them in a synchronously way too. If the queue, where the notification is posted is known, a blocking interceptor implementation can also be achieved. Therefore the method – **(void)** addObserverForName:object:queue:usingBlock: must be used, specifying the queue where to consume the notification. When trying to achieve a blocking variant with this approach, the

Listing 5.6: Code fragment to demonstrate a blocking Interceptor using NSNotification.

```

1 @implementation AppDelegate
2 ...
3 - (void)applicationDidBecomeActive:(UIApplication *)application {
4     ...
5     // Send Notification on Main Queue
6     dispatch_async(dispatch_get_main_queue(), ^{
7         [[NSNotificationCenter defaultCenter] postNotificationName:@"
            MyApplicationDidBecomeActive" object:application userInfo:
            userInfo];
8     });
9 }
10 ...
11 @end
12
13 @implementation Interceptor
14 ...
15 - (id)init {
16     self = [super init];
17     [[NSNotificationCenter defaultCenter] addObserverForName:@"
        MyApplicationDidBecomeActive" object:nil queue:[NSOperationQueue
        mainQueue] usingBlock:^(NSNotification *note) {
18         // Place interception execution here
19     }];
20     return self;
21 }
22 ...
23 @end

```

queue where the notification will get sent has to be used. A small code example of a blocking interceptor implementation with notifications can be found in Listing 5.6. In this listing a notification is posted on line 7. Due to the method call on line 6 it is established that this notification will be posted on the main thread. The interceptor is registering itself for the notification on line 17 and specifies the main queue as queue to consume the notification. Since the main queue is running on the main thread the notification will be sent and received on the main queue. Therefore the notification is consumed synchronously which represents a blocking implementation of an interceptor.

Notifications are not meant to use in such a way. The example was only given to demonstrate the possibility of a blocking implementation of interceptors with notifications. However, notifications should only be used to be consumed asynchronously and therefore are suitable for non-blocking interceptor variants.

## Objective-C Runtime

The Objective-C runtime [App13e] allows to manipulate the behavior of a method of a class during execution and provides a large set of functions to change the code during a run. One

**Listing 5.7:** Code fragment to extend the method `-(BOOL)applicationDidBecomeActive:` from the class `AppDelegate` with intercepted code during runtime.

```

1 @implementation AppDelegate
2 ...
3 // standard implementation of applicationDidBecomeActive
4 - (void)applicationDidBecomeActive:(UIApplication *)application {
5     // Something happens here
6 }
7 ...
8 @end
9
10 @implementation AppDelegate(Interceptor)
11
12 + (void)load {
13     Method originalMethod = class_getInstanceMethod(self, @selector(
14         applicationDidBecomeActive:));
15     Method interceptorMethod = class_getInstanceMethod(self, @selector(
16         intercepted_applicationDidBecomeActive:));
17     method_exchangeImplementations(originalMethod, interceptorMethod);
18 }
19
20 - (void)intercepted_applicationDidBecomeActive:(UIApplication *)
21     application {
22     // Place interception execution here
23     [self intercepted_applicationDidBecomeActive:application];
24 }
25 @end

```

way to include interceptor code fragments could be established by the technique called **method swizzling**<sup>3</sup>.

This technique allows to modify the mapping from a selector (that is the method name) to an implementation (that is the code of the method). As contrast to Objective-C categories, method swizzling allows not only to replace a method code but also make use of the original method body. Therefore it is possible to extend a method with specific code. A sample code is shown in Listing 5.7.

The code in Listing 5.7 shows, how to implement an interceptor with the Objective-C runtime. Goal of the interception code is to place code fragments at the beginning of the call `-(BOOL)applicationDidBecomeActive:`. First of all an Objective-C category on the class `AppDelegate` has to be created (line 10-24) to add a new method to this class, in this case the new method is called `intercepted_applicationDidBecomeActive`, which includes the code to include into the original method (line 19).

By overwriting the method `+(void)load` on line 12 it is possible to handle tasks before even the first object of this class is generated. Strictly speaking this method is even called on the

<sup>3</sup>Crowd-sourced documentation <http://cocoadev.com/wiki/MethodSwizzling>

very first application start, therefore this is the right place to *swizzle* the methods. Therefore a reference to the original method is retrieved on line 13 and another reference to the new method is retrieved on line 14. These references are used on line 15 to exchange both's implementation.

After this step all calls of `-(void) applicationDidBecomeActive:` will cause an execution of the code of the method `-(void) intercepted_applicationDidBecomeActive:` and vice-versa. The call on line 20 establishes the following workflow:

1. The method `-(void) applicationDidBecomeActive:` gets called, but because of the implementation exchanged, the code of `-(void) intercepted_applicationDidBecomeActive:` gets executed.
2. The interception code gets therefore executed.
3. The call of `-(void) intercepted_applicationDidBecomeActive:` will cause an execution of the original `-(void) applicationDidBecomeActive:` implementation, because the implementation of these both methods got exchanged.

Therefore this code snippet represents a code interception of commands before the original method body. To implement an *after-call* code interception the only thing to exchange is line 19 and 20. It's even possible to create interceptors which allow to skip the execution of the original method body by adding a condition to the recursive call on line 20.

## Comparison

Each of the above mentioned implementations has specific characteristics, strengths and weaknesses. This comparison summarizes which approach suits best for certain use cases.

**Notifications.** Since notifications are meant to use for asynchronous communication between loosely coupled and independent software components they are best to use when implementing non-blocking interceptors.

**Delegation Forwarding.** This method causes a tighter coupling compared to notifications. Delegation forwarding can be used to implement a blocking interceptor mechanism. Most of the time it might not be necessary to use blocking interceptors, but delegation forwarding seems to be a good solution when such an interceptor type has to be used.

**Objective-C Runtime.** The most powerful approach is represented by the technique involving the Objective-C runtime. However using this technique might cause some problems. Code introduced with runtime manipulation is hard to debug and to maintain. This technique also represents a challenging topic in the area of Objective-C programming, hence not all developers might be skilled enough to fully understand the meaning and benefit of this approach.

Listing 5.8: Example style sheet for loading with *UISS*.

```

1 {
2     "Variables": {
3         "myTintColor": "green"
4     },
5     "UITabBar": {
6         "tintColor": "$myTintColor",
7     },
8     "UIButton": {
9         "tintColor": "$myTintColmyTintColor",
10    },
11    "PHMCustomView": {
12        "phm_backgroundColor": "blue",
13        "UIButton": {
14            "tintColor": "yellow"
15        }
16    }
17 }

```

## 5.4 UIAppearance Wrapper

In Listing 4.7 an example is shown to configure the appearance of views with the *UIAppearance API*. This method allows to sum up all style configurations of views, place it in one method and is executing at the beginning of the life cycle of the application. These calls can be extracted into one style loader class, to have all the appearance related code fragments in one place. Subclassing this class may help to change the styles for each product of the software product line.

This may cause a stylesheet as code fragments which is hard to read and also compiled into the application. Robert Wijas built an open source library called *UISS (UIKit Style Sheets)*<sup>4</sup> which acts as an additional layer above the appearance API. It uses style sheets persisted as JSON files with a particular syntax, which will get translated into **UIAppearance** code during loading this style sheet file. Hence the **UIAppearance** code will be generated using the stylesheet in JSON format as input. Due to this translation, this approach is using the variability mechanism called generation.

This library makes it very easy to define styles for an application in a way inspired by CSS. An example how to accomplish the same configuration as in Listing 4.7 is shown in the JSON in Listing 5.8. To load such a stylesheet the method `+(void) configureWithJSONFilePath:` should be called after the application finished launching.

Another benefit gained by the usage of this library is the ability to create variables for styles, which can be used at several places inside the JSON. The example got extended by setting the color of the **UITabBar** to the same color as the **UIButton**s, therefore the variable `myTintColor` was created, hence it is possible to change this color in a single place inside the JSON, which lead to an improved maintainability and a single spot to change, if the color should be changed.

<sup>4</sup>UISS GitHub Repository <https://github.com/robertwijas/UISS>





# Variability Model for multi-client capable Mobile Applications

This chapter uses the the results of the requirements analysis presented in Chapter 2 to construct a feature model. The details of the mapping between these requirements and the features of the constructed feature model will be discussed and explained.

After creating this mapping, possible variability mechanisms and developed techniques from Chapter 5 will be used to explain a possible implementation of the variability of the software product line introduced by the feature model.

At the end of this chapter an insight into the configuration management is given to show how this variability of the software product line can be managed over derived products. How to keep track of different products and different versions of core assets will also be shown. Therefore this part of this chapter will illustrate how to manage variability with respect to the tools used for developing an iOS application.

## 6.1 Feature Model of the Mobile Application

In Figures 6.1, 6.2, and 6.3 a feature model covering the functional requirements from Chapter 2 is shown. The non-functional requirements are not represented as features in the feature model, since most of them are covering overall aspects regarding software quality and maintenance and therefore will not be represented as features in the model.

To enhance readability the feature model was split up and refers to two sub feature models. Figure 6.1 shows the mandatory feature *File Browsing* which is described in more detail in Figure 6.2, the mandatory feature *Sync Engine* is described in Figure 6.3.

### 6.1.1 Overall Feature Model

The **File Management** feature is represented as an optional feature group with its sub-features **Saved Search**, **History**, **Favorites** and **Custom Collections**. All these features should help the

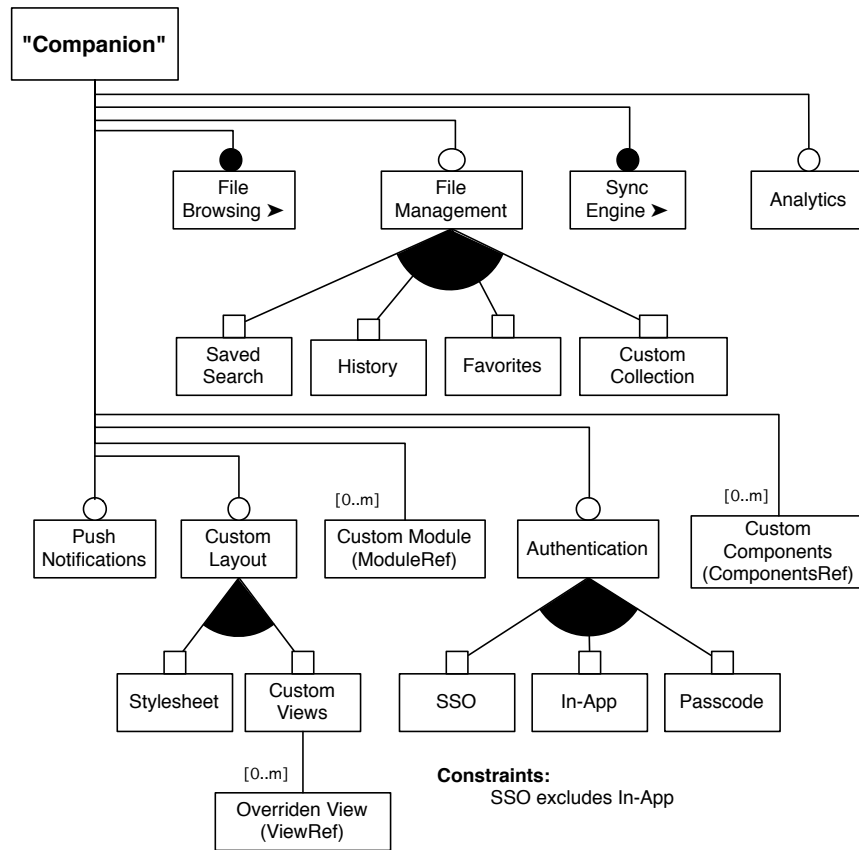


Figure 6.1: Overall feature model covering requirements after analysis.

user to manage the user's files in a more flexible way. The user will not have to use the folder structure given from the file system of the synchronization server but will be able to manage the files and folder by himself.

The application should allow to change the style by applying an optional stylesheet to allow to design the application with respect to the user's needs. If a layout change is necessary that can not be expressed with a stylesheet the user can use the feature **Custom Views** to specify views to substitute given views. Hence this allows heavy user interface customization if necessary. This feature can be applied to several views, therefore a reference to the substituted view has to be provided. These layout features are grouped into the feature group named **Custom Layout**.

To authenticate the user against the used sync service or to another registry service an optional **Authentication** group is present. Users may be authenticated through an in-app login, single sign-on, or a simple passcode where the choice of a single sign-on excludes the ability of an in-app login. This restriction is modeled with a constraint which restricts the choice of an authentication mode.

To extend the application with customer specific requirements which can not be applied to

other customers the feature model provides a method to specify additional **Custom Modules**. This feature allows to keep the feature model more flexible as well as to introduce features that may come into being in the future. When such a custom module will be used in more products later on the feature model will be extended. In such a case this **Custom Module** will be included into the core assets.

If a heavy customization of the application is necessary **Custom Components** can be attached to the application which may act as substitute for other components or as stand alone components as well. Similar to the above mentioned modules such a custom component can be introduced to the core assets when used in more products than one. Therefore, the substituted component will mark a new variation point of the software product line. The substitute (or the different substitutes) will be present as variants of this variation point.

Custom modules are acting as independent and capsulated modules for the application, whereas custom components are smaller components that interact with other components. Custom components can also be used as substitutes of given components, whereas custom modules are not used for substitution of other modules.

The feature **Analytics** is represented as optional feature. This feature allows to track the application's activity to a tracking service.

To establish a communication channel to the application the optional feature **Push Notifications** has to be selected.

### 6.1.2 Sub-Feature Model “File Browsing”

The feature **File Browsing** shown in Figure 6.2 supplies different views, i.e. list view, grid view and presentation mode. At least one **View Mode** feature has to be selected for file browsing, otherwise the files could not be presented inside the user interface. The different view modes are modeled as features of this feature group.

The group **File Actions** handles the available actions on files, i.e. to control if a file is allowed to be printed, opened, send via mail, tagged or commented.

Another way of accessing folders is introduced by the feature **Shortcuts**. This feature allows the user to quickly jump to a particular folder in the folder hierarchy without browsing through the whole folder tree.

### 6.1.3 Sub-Feature Model “Sync Engine”

Figure 6.3 describes the features of the file sync engine in more detail. The type of sync is represented by the feature group **Sync Type** and specifies the technique which should be used to retrieve data from a sync engine. Therefore the group specifies whether the underlying sync service is working in a recursive way, provides full list access, or delta sync mechanisms. Exactly one feature of this group has to be selected to allow the application to synchronize with a sync provider.

How often the application will synchronize with a provider and therefore update, add or delete files and folders is handled by the group **Sync Frequency**. The different features of this group define the possible moments such a sync will be triggered. At least the sub-feature **Manually** has to be selected to allow the user to manually trigger a synchronization process.

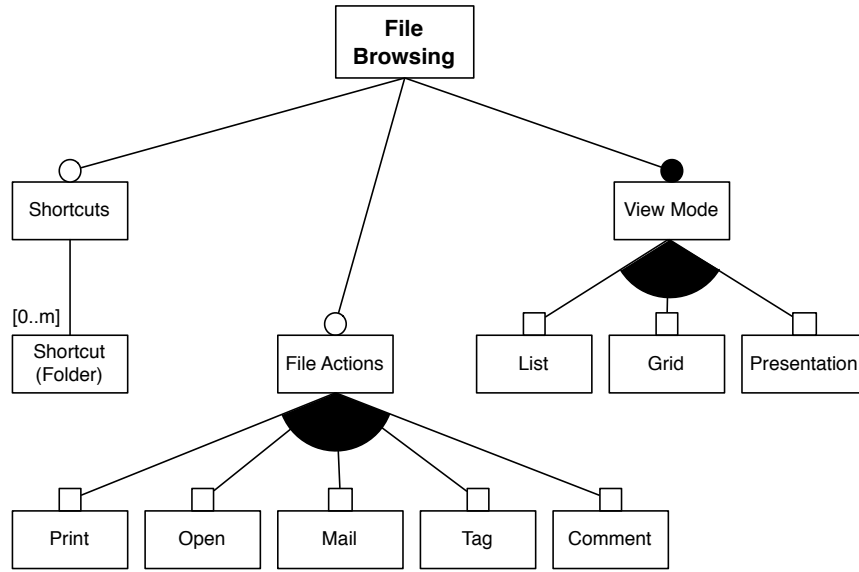


Figure 6.2: Sub-feature model representing file browsing requirements.

Otherwise the user would have to wait until the application will be reopened, freshly started or a predefined time interval has succeeded.

Files can be downloaded **On Demand** or automatically during sync (the so-called **Replication Mode**). When downloading all files during the synchronization process the file storage usage will increase and the synchronization process in general will be slower. When downloading files on demand the user will not have all the files available offline which may lead to some issues in some use cases. Therefore the user should decide which **Download Type** may be suitable. Another way of defining the download type is provided by a constraint-based type (represented as **Custom** download type). There are two ways to constrain this custom download type:

1. *Whitelisting*: When using a whitelist the custom download type is acting like the download type *On Demand*. The whitelist specifies rules to determine which files should be downloaded during sync.
2. *Blacklisting*: When using a blacklist the custom download type is acting like the download type *Replication Mode*. The blacklist specifies rules to determine which files should not be downloaded during sync.

Therefore the rules or constraints describe exceptions when using the download type *On Demand* or *Replication Mode*. The constraints can be presented as a file size limit or a list of file extensions.

Exactly one feature of the feature group **Sync Provider** has to be selected. These features are represented as sync providers and are responsible to establish a connection to a server to

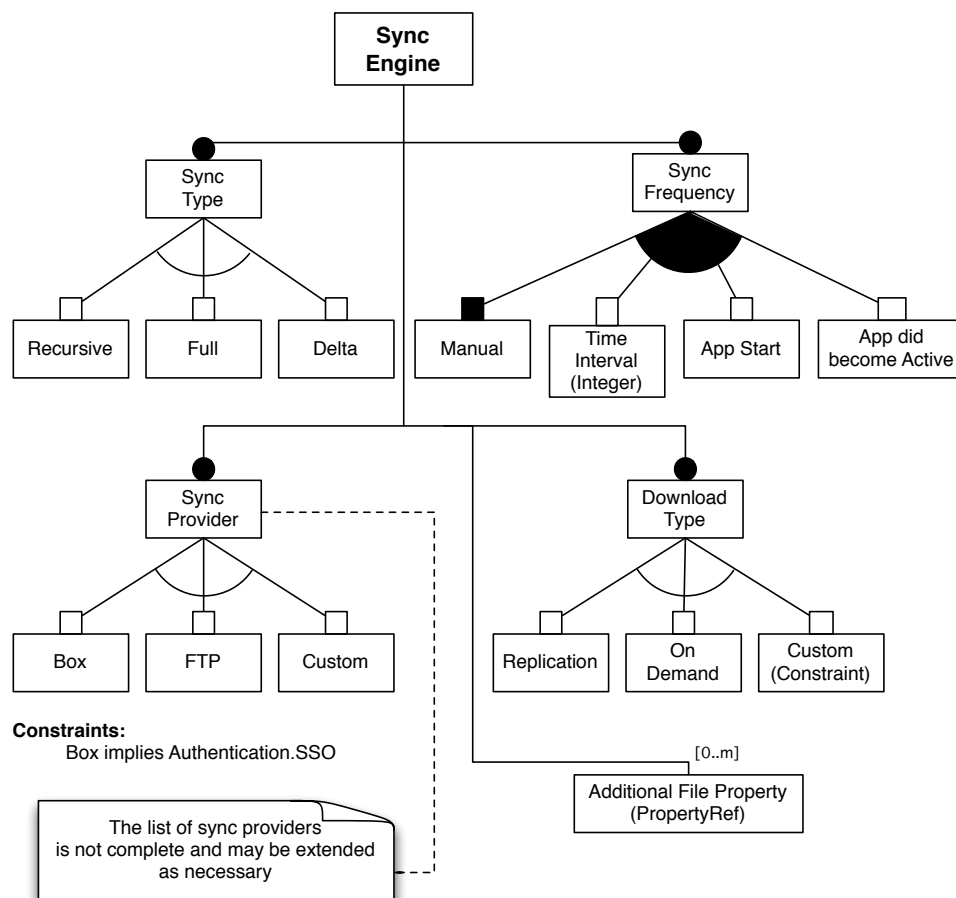


Figure 6.3: Sub-feature model representing sync requirements.

allow a synchronization process with the application. The list of sync providers in Figure 6.3 is just an excerpt of all supported sync providers and may be extended as new customers with new custom service solution are requesting to add their service. The choice of a sync provider can influence other features. In the given example the sync provider Box implies the choice of a single sign-on, since the Box API does not allow other authentication methods. The choice of the sync provider also may influence the choice of sync type depending on the result type of the API response.

Some sync providers may have additional properties that are not included into the data model. When a user is requesting these properties to be synced to the application the feature **Additional File Property** allows to extend the given data model. A description of the property has to be given to identify the additional file property.

## 6.2 Mapping Features and Variability Mechanisms

Table 6.1 shows the mapping from requirements to features of the feature model and also a mapping from that features to variability mechanisms. The mapping is explained in more detail in Section 6.3 and describes, how the different features get included into the application as well as how these features can be configured.

Requirement	Feature	Variability Mechanism
File- and folder-structure syncing	Sync Engine - Sync Type	Class Loader and Configuration Loading
File download on demand, replication or custom	Sync Engine - Download Type	Configuration Loading or Class Loader
Tags, Comments	File Actions	Configuration Loading
Additional Meta Information	Additional File Property	Transformable (Core Data)
Favorites Custom collections History Saved search	File Management	Module Handling
Passcode Authentication	Authentication	Delegation Forwarding, Configuration Loading, Interceptor
Timeline	Custom Module	Module Handling
Notifications	Push Notifications	Compiler Directive and Delegation Forwarding
User interface customization	Custom Layout - Stylesheet	UIAppearance-Wrapper (UISS)
Custom user interface elements	Custom Layout - Custom Views	Class Loader
Custom modules	Custom modules	Module Handling
Custom components	Custom components	Class Loader
Analytics Service	Analytics	Interceptor

Table 6.1: Mapping from requirements to features and from features to variability mechanisms.

## 6.3 Realization of the Mapping

This section describes the realization of the mapping using Objective-C and the established methods from Chapter 5. Before creating this mapping, another concept was developed to introduce variability into the user interface to allow to extend this user interface in a flexible way. This method is described as follows.

### Module Concept

The module concept was developed during this thesis to manage variability inside the user interface and to allow to design an application which can be extended with certain independent functionalities. For these independent functionalities the term module was introduced.

This concept uses an often used user interface paradigm that can be found in several mobile applications like Facebook, Twitter and several apps from Google. This paradigm is illustrated in Figure 6.4 and divides the user interface in three different parts.

1. **Content Area:** The content area is displaying the main user interface of a selected module. The user is interacting with the application most of the time in this area and therefore this screen is using the whole dimension of the screen
2. **Left Drawer:** The left area can be revealed with a button which is located at the left upper corner of the screen or with a swipe gesture from the left border of the screen. This area contains a list of all modules that can be presented in the content area. Triggering one of these modules will hide the left area and update the content area with the user interface provided by the selected module. The modules are managed in this area in a table view which allows to extend the list of modules very easily without disrupt the user interface's presentation.
3. **Right Drawer:** The right drawer can be used for a reduced representation of a module. For example, Facebook is using this area to present the chat module. This area allows to quickly access a certain functionality of the app with a single interaction. To reveal this area a button is provided on the right top corner of the screen or with a swipe gesture from the right border of the screen. If more than one module should be represented in this area a way has to be found to easily configure which one of these modules should be shown. This can be established with a long press gesture on the button which will show a dialog where the user can pick the module to see. The last picked module will be triggered afterwards when revealing the right drawer again.

To configure the appearance of different modules a class concept was designed which is shown in Figure 6.5. The `CPNModuleProvider` holds a list of modules which can be presented inside the left drawer and a list of mini modules which can be presented in the right drawer. At the applications start this provider loads the modules from two different configuration files. These configuration files contain the names of the modules (to be precise the class names of the modules). The file *Modules.plist* is responsible for the entries loaded into the property



Figure 6.4: Sketch and interaction description of an user interface handling multiple modules.

`modules` whereas the file *MiniModules.plist* is responsible for the entries loaded into the property `miniModules`.

A module itself has to subclass the class `CPNBaseModule`. This base class is offering different properties which are necessary to retrieve the designated view controllers for the different areas. The property `moduleViewController` is used to load the screen from the left drawer to the content area of the screen. The name which will be displayed inside the left drawer is stored in the property `moduleName`.

The `miniModuleViewController` property is providing the user interface that will be shown in the right drawer when revealing this area of the user interface.

Another extension to the above introduced concept is the ability to have subitems in a module. Lets assume a module is provided that displays the current weather of a city. Therefore a module can be created with a view controller to display in the content area and the name *Weather*. When extending this feature in a way to have more than one city a weather can be displayed for the entry in the left drawer should be expanded listing all available cities as subentries of this



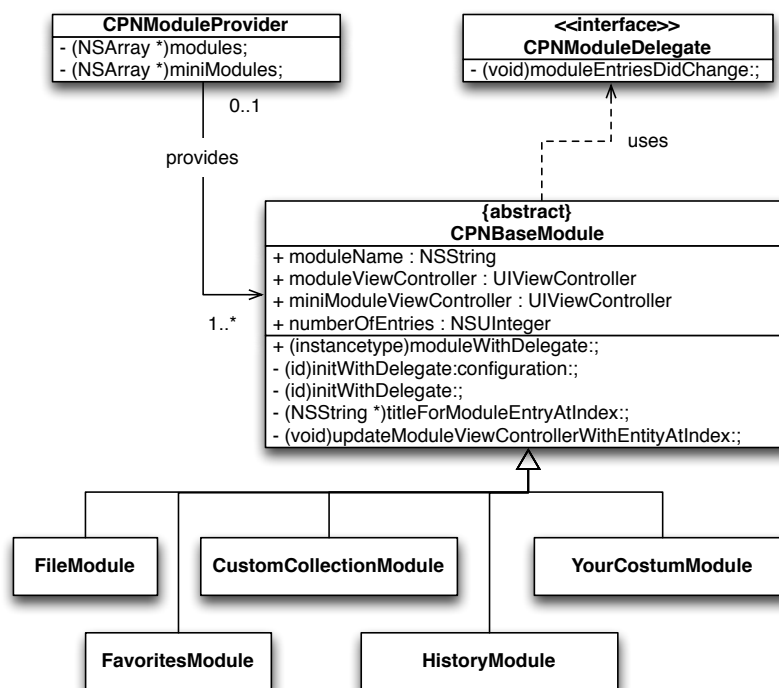


Figure 6.5: Class diagram of the *Modules* concept with several example modules.

module. After tapping on a city the content screen will be loaded with the selected city. Therefore three more methods are defined inside the `CPNBaseModule`. The property `numberOfEntries` is managing the list of subentries of the module, the method `-(NSString*)titleForEntryAtIndex:` is providing the name of the subentry which should be displayed in the left drawer. In case another subentry is already displayed in the content area the method `-(void)updateModuleViewControllerWithEntityAtIndex:` will update the view controller with the new selected subentry.

The left drawer also has to be informed that the numbers of subentries had changed to update the user interface. Therefore a module holds a reference to a delegate (in that case the left drawer) which will be informed if a change occurs that influences the number of subentries.

To configure a module the `CPNBaseModule` is designed to be used with the *Configuration Loader* mechanism. Therefore the module can be configured using a configuration file, e.g. the weather module can be loaded with a different set of cities for each derived product of the software product line. Therefore the configuration file should hold a list of cities that will be loaded into the weather module.

### 6.3.1 File Browsing and File Management

When keeping the introduced module concept in mind the features of the feature group *File Management* can be presented as modules. The module *File Browsing* therefore is acting as mandatory module which every product of the software product line should contain. The modules *Favorites* and *History* are optional ones and could also be presented as mini module in the right drawer of the user interface. *Custom Collection* and *Saved Search* are modules that can be integrated in the menu inside the left drawer and can also have subentries for each of the custom collection and each of the saved search query.

### 6.3.2 Custom Modules

Custom modules can also be handled by the *Module Concept* described before. Creating a custom module is handled by creating a new subclass of `CPNBaseModule`. To integrate this new module into an application the *Modules.plist* file and/or *MiniModules.plist* has to be extended with the name of the class of the new custom module. If such an entry is present the `CPNModuleProvider` will load this new custom module for the derived product of the software product line.

### 6.3.3 File Actions

The `Configuration Loader` can be used to enable specific file actions for the application. These file actions are represented as flags which are managed by one single subclass of `CPNGlobalConfiguration`. This class is also used to store API tokens for the sync provider, the over-the-air distribution system and the crash reporting tool. These tokens should not be provided by a configuration file since resources like images, configuration files, icons, etc. can be extracted from the iOS application's bundle which may cause a security issue.

The class `CPNGlobalConfiguration` should be subclassed for each product of the software product line to set all necessary properties for the application.

### 6.3.4 Sync Engine

The *Sync Engine* is split up into several classes to allow configuration and variability. An class diagram representing the class concept is shown in Figure 6.7, the used mechanisms to establish variability and to meet the requirements are described below.

#### Sync Type and Sync Provider

The application is communicating with one concrete subclass of `CPNBaseSyncEngine` and only uses the provided methods from this base class. The *Class Loader* concept is used to select the accurate subclass, i.e. one of the concrete classes `CPNRecursiveSyncEngine`, `CPNDeltaSyncEngine` or `CPNFullSyncEngine`. Each of these concrete subclasses represents one sync type. As described in Figure 6.7, an instance of a `CPNBaseSyncEngine` is using exactly one instance of a subclass of `CPNBaseSyncService`. This services conforms to a sub protocol of `CPNSyncServiceProtocol` to ensure that every method required for the selected type of sync

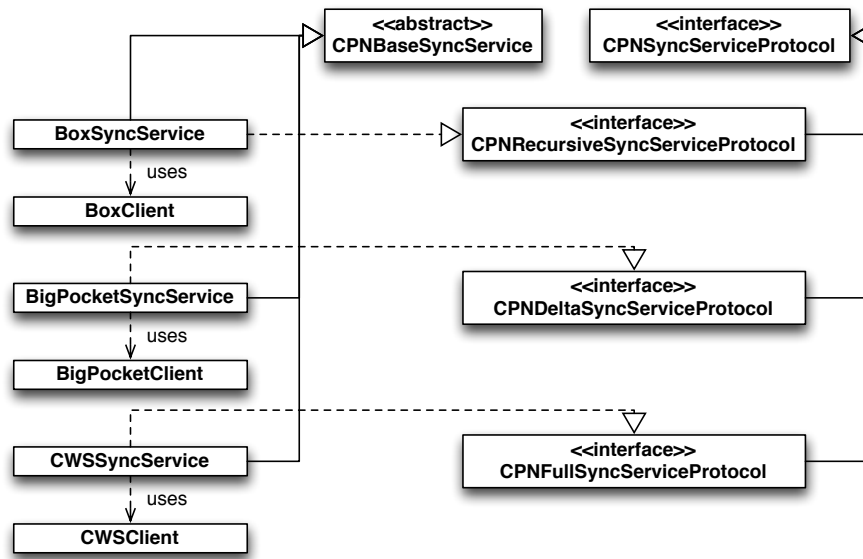


Figure 6.6: Class diagram of a completed *Sync Engine* with concrete subclasses of *Sync Service*.

mechanism is provided. Each type of sync engine has its own protocol, hence requesting files is varying from each of them.

The concrete subclasses of `CPNBaseSyncService` may use a third party library to communicate with the specific server. As an example, a sync service implementing a synchronization to Box may use the third party library from Box. The selection of the sync services is also handled by the *Class Loader* by specifying which concrete subclass of `CPNBaseSyncService` should be used as substitute.

To show how the relationship between the protocols and the concrete subclasses with its clients can look like an example is provided in Figure 6.6. This example complements the before mentioned class diagram with three sync services. The first service access data from Box, another service retrieves huge amount of data from a delta sync service. The third service provides a single call to retrieve a whole list of files and folders with its modification date and other useful information. The example also illustrates that each of the sync service has its own client library. This library handles the communication to the server and also shows that corresponding to the type of sync service the service has to conform to a specific protocol. It is also possible that a sync service conforms to multiple protocols, therefore this service could be used for different types of sync engines.

### Download Type and Sync Frequency

Variation points which can be expressed with properties of classes, are using the *Configuration Loader* concept. At the beginning of the life cycle of a sync engine properties like sync fre-

quency and download type are loaded from the corresponding configuration. To simplify the configuration loading for the used sync service the configuration of the sync engine is passed to its service (*Configuration Forwarding*), which consumes its values, like server URL, predefined username and/or password, and so on. While extending the properties of sync engines and sync services name clashes have to be kept in mind, otherwise it will end in an unexpected behavior during the property initialization through the configuration file.

The enumeration object for the *Download Type* provides values for the *On Demand* and *Replication Mode* download type. The property which determines which download type the sync engine should use can be set with *Configuration Loading*. The third download type *Custom* from the feature model is missing in that enumeration.

This custom download type is implemented as subtype of the two other download types. Additional properties are defined to configure these two download types. The approaches to configure the download types with constraints are the following:

1. Filtering files by file size: By setting the `maximumSizeForDownloadMechanismReplicaton` property through the *Configuration Loader* files will be downloaded automatically, if the file size is lower than the value of this property. These constraint will only have an effect when the download type is set to *Replication Mode*.
2. Filtering files by its file extension: By setting the array `fileExtensionConstraintList` a list of file extensions can be provided which will be used to filter the files that should be downloaded. When using the *On Demand* download type this array works like a whitelist, hence all files with a file extension contained in that list will be downloaded. When using the *Replication Mode* this array works like a blacklist, hence all files with a file extension contained in that list will not be downloaded.
3. Filtering files with more complex constraints: To filter files for downloading with more complex constraints subclassing as variability mechanism should be used. More complex constrains can be implemented to decide if a file should be downloaded during sync or not when a sync engine gets subclassed.

### Additional File Properties

Various sync services may offer a set of additional meta information, like comments, tags, the file creators name, a description, notes, etc. To allow a variable data model to save this information to the file object a schema free or at least schema tolerant way to persist data had to be found.

Apple's framework *CoreData* [Zar13] was used to solve this problem. *CoreData* does not allow a schema free data model but lets the developer define an attribute of type `Transformable` [App13b]. When defining an attribute with this data type the developer also has to specify a class, which is responsible for transformation of this property from `NSData` (the actual type of the property inside the database) to another data type. To implement a schema free addition to the schema of a file object such a property was defined to transform the data from `NSData` to `NSDictionary`. This additional property in dictionary format can be used to add any number of additional properties to a file.

A negative side effect when using this approach is the lack of query possibilities. It is not possible to setup queries which allow filtering on properties stored in that dictionary.

### 6.3.5 Authentication

As shown in Figure 6.6 the sync engine provides a method to authenticate a user with an username and password. The user interface can be configured using the *Configuration Loader* to switch between a standard login or passcode authentication. Single sign-on on mobile application is working with a redirect to the service providers website which handles the authentication. The user will be redirected to the application with an URL redirect after success. This application flow makes it hard to implement the authentication mode with the concept of configuration loading.

One solution to solve this problem is to use subclassing. The applications delegate has to be subclassed to handle the URL redirect and therefore the authentication. Since the fact that the application delegate marks an entry point for the application life cycle, it seems not elegant to subclass this important class for authentication purposes. Instead *Delegation Forwarding* will be used to inform another class about the occurrences of these authentication events in a blocking way.

### 6.3.6 Analytics

To support different analytics services an approach with non-blocking interceptors was implemented with notifications. First of all, the events which might be interesting for an analytics service had to be examined and specified. On every occurrence of such an event a `NSNotification` is posted from the application to the `NSNotificationCenter`. The notification itself should have a meaningful name and useful additional information attached. For the event which will occur on every startup of the application this additional information can be presented as a counter which determines how often the application has already been launched.

A class called `CPNAnalyticsHandler` is provided to register itself as observer for all these notifications. For each notification a method of this handler will be called. To implement a handler for a analytics service (like Google Analytics<sup>1</sup>, Localytics<sup>2</sup> or Flurry<sup>3</sup>) a subclass of this handler should be created. This subclass can react on events by overwriting the provided methods. With this method it is easy to integrate multiple analytics services, reacting on different events.

### 6.3.7 Custom Components

Custom components are handled with the *Class Loader* concept. For example a class `MySubstitute` can be used instead of the class `OriginalClass` by extending the *Class Loader's* configuration file *OverriddenClasses.plist* with an entry with the key `OriginalClass` and the value `MySubstitute`.

---

<sup>1</sup><http://www.google.com/intl/en/analytics/>

<sup>2</sup><http://www.localytics.com>

<sup>3</sup><http://www.flurry.com>

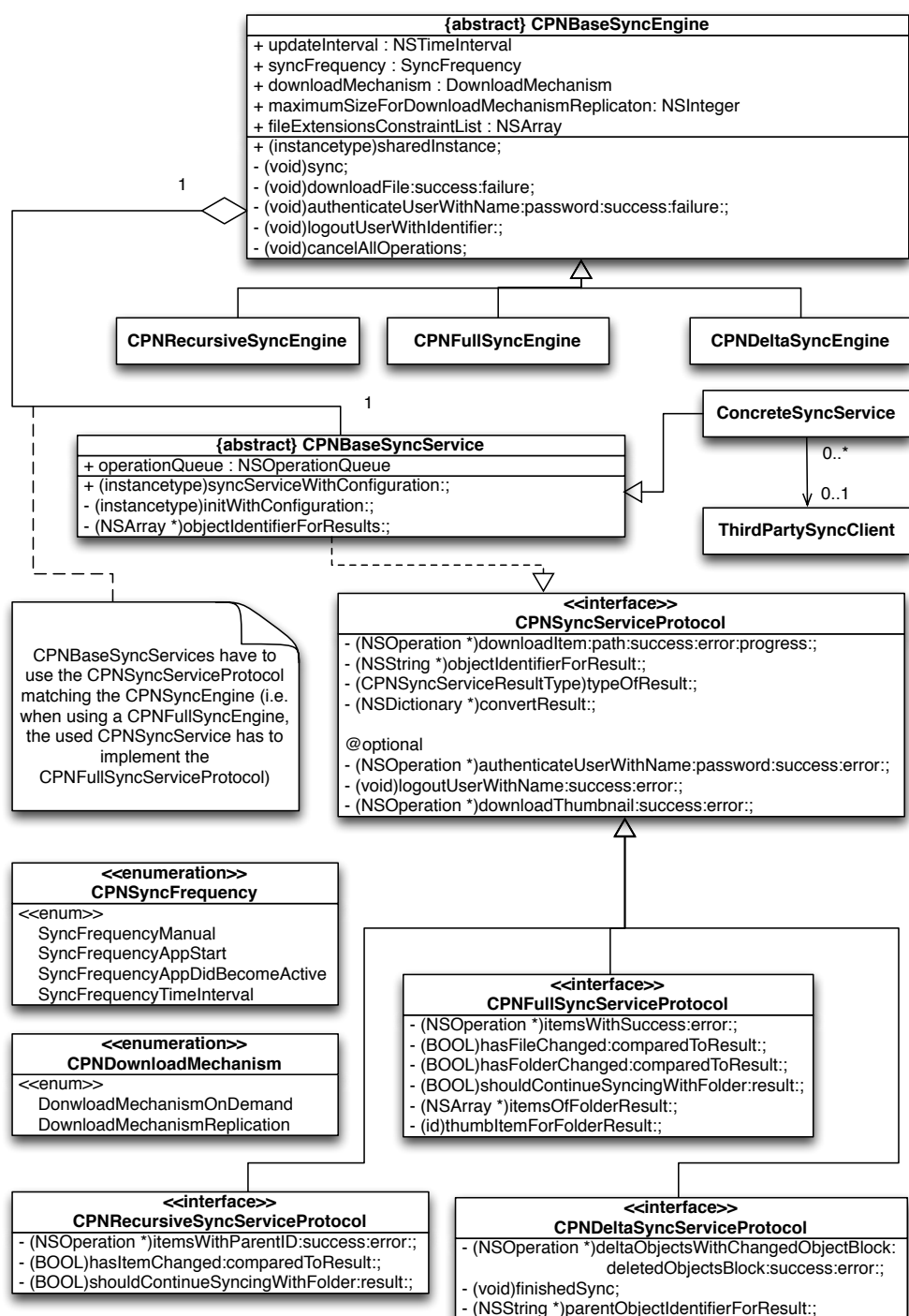
### 6.3.8 Layout

Stylesheets are handled with the *UIAppearance Wrapper UISS*. For each product of the software product line a new stylesheet can be applied. This stylesheet can change the appearance of views which properties are exposed for the *UIAppearance API*. An iOS application usually contains several images that are used as background images or icons. A disadvantage when using images comes into being because of the fact that images cannot be tinted and therefore the style of this images can not be adopted to meet the design requirements. As a solution to this problem all icons are drawn using *CoreGraphics*, a Framework from Apple to use the graphics engine directly. Because of the usage of *CoreGraphics* the icons are acting like vector-based images which will reduce the applications memory fingerprint. Due to the drawing a style can be applied to the icons. To avoid boilerplate code inside the stylesheet *UISS* introduced variables for colors, borders, shadows etc. These variables can be used at several places inside the stylesheet. Hence changing colors that are used from several user interface elements is easier when a variable is defined for that inside the stylesheet.

Some design requirements can not be fulfilled with styling a view. When heavy customization for views of an application is necessary it might be a better solution to create subclasses for these views. These subclasses can implement the required design and can be registered as substitutes of the standard user interface elements by using the *Class Loader*. Hence these views are acting like *Custom Components*.

### 6.3.9 Push Notifications

The feature *Notification* is used to send out messages to all user of an application. Apple introduced a service called *Push Notifications* to establish a communication channel to the user. To integrate this service the application's delegate has to be modified to register for this service and to react on incoming notifications. To prevent an overloading of this delegate the mechanism *Delegation Forwarding* can be used. Therefore the code snippets which are necessary to integrate Apple's *Push Notifications* can be included into the subclass of `CPNGlobalConfiguration`. Some code snippets that can not be outsourced to this configuration class can be enabled or disabled using compiler directives. Therefore code will be excluded for products which may not use this feature.

Figure 6.7: Class diagram of the *Sync Engine*.

## 6.4 Configuration Management

When it comes to multiple products of a software product line configuration management becomes a key aspect to maintain the different versions. This chapter describes the configuration management which was used to implement the software product line using Xcode as a development tool and Objective-C as the programming language.

First of all, it will be explained how the individual parts of the software product line are split up within the Xcode project. Furthermore it will be shown, how this is reflected in the used source control management GIT.

The individual meanings of the files are described afterwards. It is shown which files are responsible for the configuration options and how these files work together. This description will be used later in Section 7.1 to show how a product of the product line is derived.

### 6.4.1 Project Setup

All files that have to be changed or are only used for one specific product are separated into an own static library. This library (called *CompanionConfiguration*) is saved into an own GIT repository and managed as submodule to have even a better separation to the core assets and code base.

To keep track of the versions of the code base and the product, Xcodes build number and version number are used. The build number is used to mark the version of the code base, which will be incremented on every beta or release build of a product of the software product line. The version number is part of the configuration submodule and will be increased semi automatically. If managing a new release, the build number is set (e.g. to 2.0.1). While creating beta releases of the application for this specific product the version number is created by appending a “b” and a two digit number which will be increased for every beta release (e.g. 2.0.1b01, 2.0.1b02).

Each application must have its own bundle identifier which is represented as string in reverse DNS-format [App13a]. This identifier will be automatically extended by the keyword “stage” for beta releases. Therefore the release and beta version for the application will have different bundle identifiers, which leads to the ability to have both versions installed on the same device for testing purposes.

To keep track of different versions and products a GIT branch is created inside the *CompanionConfiguration* submodule for each product of the software product line. Additionally the commits are tagged after a release with the corresponding branch name and version number to easily switch back to this version.

### 6.4.2 File Structure

The file structure in the project navigator is shown in Figure 6.8. The group *Companion* represents the code base and core assets with all its standard view controller, views, data models, etc. The file *Info.plist* (Figure 6.8, Mark 1) is used in an iOS application to configure the main aspects of an application, like supported interface orientations, bundle identifier, name of the application, paths to icon files and URL schemes. This file is also located in the main GIT repository, the properties to change are outsourced to project configuration files. These files



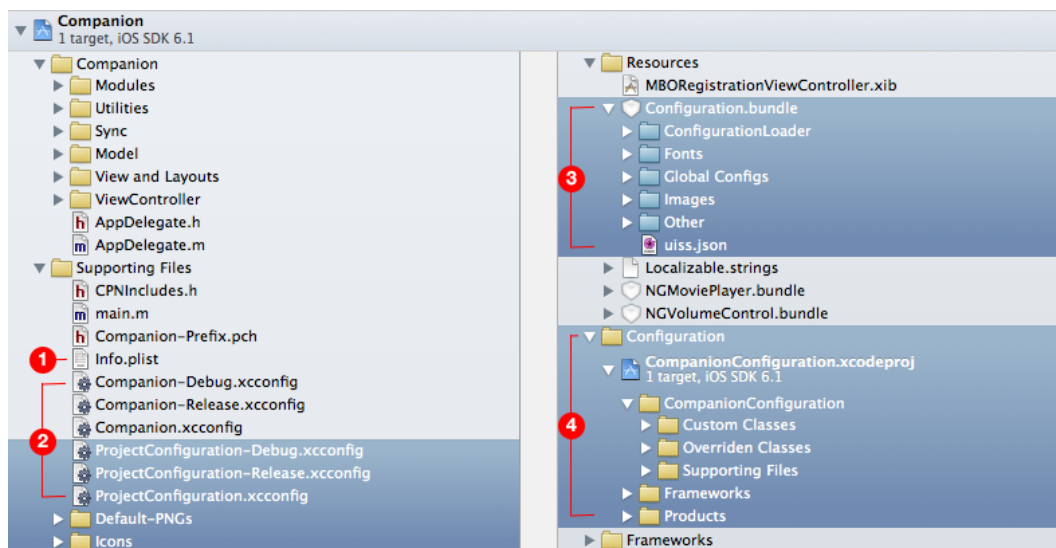


Figure 6.8: Project setup and file structure inside the project navigator.

have the extension *xcconfig* (Figure 6.8, Mark 2) and contain properties which can be used to configure the Xcode project.

The configuration files are separated into two groups. *Companion.xcconfig* stores the build number and other informations that are common for all products of the software product line and therefore belong to the core assets. *Companion-Release.xcconfig* and *Companion-Debug.xcconfig* are handling the bundle identifier (which will be derived from the config file *ProjectConfiguration.xcconfig*) for live and beta versions (with the appended string “stage”).

The blue marked resources are stored in the above mentioned submodule *CompanionConfiguration* to handle all product specific files and sources. The *ProjectConfiguration.xcconfig* files are managing the name of the application, the bundle identifier, the URL scheme and other relevant information. The file path to the icons and default PNGs (which handle the start image for iOS applications while loading the app from memory) are hardcoded inside the *Info.plist*, the images itself vary for each product. Typical resources of the static library are embedded as a resource bundle called *CompanionConfiguration.bundle* (Figure 6.8, Mark 3). Inside this bundle product-specific images, fonts and stylesheets are stored as well as configuration files in plist format to handle configuration for the *Class Loader*, *Configuration Loader* and *Module Provider*.

The last part of the submodule is the static library itself (Figure 6.8, Mark 4). It contains all source code files which are not part of the code base but necessary for the product of the software product line. This includes custom classes, like custom modules, custom views, etc. as well as all overridden classes which may be used by the *Class Loader*.



## Evaluation and Discussion

In this chapter, chosen design decisions are evaluated using the descriptive design evaluation method [HMPR04]. For this evaluation method detailed scenarios are constructed around the artifacts to demonstrate its utility in combination with the design decisions. The requirements for the used scenarios for the evaluation were derived from real customer queries. All customer requirements of the scenarios should be implemented without making monumental changes on the core assets. The evaluation itself is divided into two parts for each scenario.

- **Domain and Requirement Description:** Description how the application will be used from the customer, which persons actually will interact with the application and which goal they are following by doing it. The customers requirements will be summed up including a mapping to features of the feature model.
- **Customization Details:** Detailed description how the features were included and described steps which were necessary to become an acceptable output represented as product of the software product line. Following metrics were collected during implementation
  - Changes in Code Base, which were necessary to implement a certain feature
  - Number of configuration files, which were created for the *Configuration Loader*
  - The number of overridden classes describe the numbers of classes, which will be loaded from the *Class Loader*
  - Number of new classes, which were created especially for this product during implementation
  - Number of additional resources like images, fonts, etc.
  - Number of lines in the stylesheet to meet the design requirements

Prior to this evaluation, the steps that are necessary to create a starting point for the product derivation are described. These steps are required for each product of the software product line and are similar for each product.

At the end of this chapter a critical analysis is made to identify possible improvements and problems to improve the design decisions.

## 7.1 Product Derivation

To derive a product from the software product line the same steps must be performed every time. This procedure provides a starting point for the product and other adjustments and modifications. These steps are explained in detail as follows.

1. A new branch of the *GIT* submodule *CompanionConfiguration* has to be created for the product. The name of the branch should represent the name of the new product.
2. Inside the static library of the new branch a new subclass of *CPNCompanionConfiguration* has to be created. This is the place to set specific file actions and provide tokens for the over-the-air distribution system and crash reporting module. This class will also be used for the *Delegation Forwarding* pattern. Hence the configuration singleton should react on events called by the application's delegate, i.e. to implement push notifications.
3. The configuration singleton class has to be registered in the *OverriddenClasses.plist* configuration file for the *Class Loader*.
4. Another subclass to register in this configuration file is the class of the sync engine which has to be used for this product and the designated sync engine. For instance, if a recursive sync should be provided with the Box sync service the configuration file should be extended by two entries. One entry with the key *CPNBaseSyncEngine* and the value *CPNRecursiveSyncEngine*. The other entry is represented with the key *CPNBaseSyncService* and the value *CPNBoxSyncService*.
5. Inside the folder *ConfigurationLoader* of the resource bundle all the configuration files for the classes have to be provided (i.e. *CPNBaseSyncEngine.plist* to set up the sync engine).
6. The file *uiss.json* of the resource bundle contains the style sheet for the application and has to be adopted to meet the design requirements of the customer.
7. Inside the folder *Images* the logo of the company, for which the product will be built, and other related images should be saved.
8. The start images and icons for the product should be replaced. They can be found inside the folder where the static library is saved. There is no need to adjust the *Info.plist* where the paths for the application's icons and start images are saved since the paths remain the same.
9. The *ProjectConfiguration.xcconfig* files have to be setup for the right application name and bundle identifier.
10. The configuration file for the *ModuleProvider* should be modified to provide the class names for the modules to integrate. If the configuration file *Modules.plist* is not present the modules for file browsing and custom collections will be used automatically.

## 7.2 AESchreder Austria

### 7.2.1 Domain and Requirement Description

AESchreder<sup>1</sup> is an internationally acting company from Austria with departments in France and Belgium. The company is selling lightning equipments for streets, public places and shopping promenades to cities and big companies. AESchreder wanted an iPad application for all employees to share product information like pictures, advertising movies, product catalogues and pictures from their equipment used in different cities. This product information presented inside the application will be used in front of their customers during selling processes and customer pitches.

Another use case of the application is represented by having all company relevant files loaded into the application, including financial documents and presentations, as well as technical specification files for a technical software used by their lightning installer. Not every employee should be able to see all the files, therefore a login has to be provided which will be used to sync the files and folders for which the employee is authorized.

The company is not using a file service at the moment and was therefore requesting a server solution where they can easily upload files that will get synced to the application. The sync service got implemented within a separate project with a web-based management interface to create new users and associate user groups. Each user group has files and folders assign to control the access. A REST API was provided to allow a synchronization with the application.

The application should download all the files on demand to reduce the memory fingerprint of the application. The user interface should be styled with the companies corporate identity. Folders should not have the standard folder thumbnail provided by the application but have a customer designed icon as thumbnail based on the names of the folders. This demand was another design requirement which should be included into the application.

The above mentioned files used by another application are not meant to be viewed from inside the application since the iPad itself is able to display these files anyway. Hence it should be possible to send these files instead of viewing it from within the application. Table 7.1 sums up the requirements of the customer and maps them to features of the feature model.

Requirement	Feature
Custom Sync Engine	Sync Engine - Sync Type
File download on demand	Sync Engine - Download Type
Restricted open action	File Actions (not possible)
User Login	Authentication
User interface customization	Custom Layout - Stylesheet
Custom placeholder images	Custom components

Table 7.1: Overview of requirements and features for AESchreder.

---

<sup>1</sup><http://www.aeschreder.at>

### 7.2.2 Customization Details

To implement the product for AESchreder the common steps have been executed to create a new configuration for this product (see Section 7.1).

First of all a sync service had to be implemented to consume the provided REST API. Since the REST API is answering with a full list of files and folder which the user is allowed to view and download, the type of the sync engine was set to “Full”. A sync service which is using the implemented sync service had to be created by subclassing the `CPNFullSyncEngine`. Properties like server URL and login behavior were configured using the *Configuration Loading* mechanism. The login process got slightly adopted to react on access right changes. Therefore a sync will be forced if the user’s groups got changed to prevent the application to display data which the user is not allowed to see.

A design was created with respect to the corporate identity of the company. Afterwards the design was implemented with a stylesheet for the application. The folder placeholder which are based on the name of the folder could not be implemented with the stylesheet. This feature request was implemented as a custom component and resulted in a heavy customization. The class `CPNPlaceholderProvider` is used inside the application to retrieve a placeholder image for a specific file type. Therefore this provider is also responsible to specify the placeholder for a folder. A custom component was created to replace this class. This replacement is delivering the placeholder image by checking not only the file type, but also the name of the file and folder. Therefore each folder can have its own placeholder image. A list which associates folder names to placeholder images was created for this new placeholder provider.

The application is providing a property which describes the allowed actions on files, hence this property specifies if an action is allowed on files or not. AESchreder requested a more fine-grained control over these actions therefore the mentioned method to just enable or disable a certain action was not appropriate anymore. Methods were created inside the `CPNGlobalConfiguration` class which evaluate to `true` or `false` to decide if an action is allowed for one specific files but not all files in general. One of these methods has the signature `-(BOOL)action:allowedForFile:`, gets an action and a file object as input parameter and answers with a boolean value. Methods to allow a fine-grained control for actions on files and tags were also created for future developments. The application is not checking the file action property directly anymore but is using this method to check if an action is allowed to execute with a given file. In case of AESchreder the method evaluates to `false` for all files that are meant to be sent instead of viewing for the action *open*. A summary of the required steps and the resulting effort is given in Table 7.2.

## 7.3 Anonymous Furniture Department Store

### 7.3.1 Domain and Requirement Description

This furniture department store is located in Europe and one of the biggest in the world. Due to legal issues, it is not allowed to mention the name of the company in this thesis. Its marketing department decided to buy a product of this software product line to have all the files on their tablets during their meetings, field work, or at home. Therefore, this product will only be used

Category	Description / Metric
Changes in Code Base	Small changes regarding instantiating of <code>CPNPlaceholder</code> to retrieve class from <i>Class Loader</i> . Additional support for more fine-grained file actions by extending the <code>CPNGlobalConfiguration</code>
Number of Configuration Files	4
Number of overridden Classes	2
Number of new Classes	1 (and one Class per Placeholder Image)
Number of additional Resources	4
Number of Lines for Stylesheet	436

Table 7.2: Overview of effort for product for AESchreder.

inside the company from the employees without interacting with the application in front of a customer. The application will be used as a pure communication tool within the department. The major requirement and most challenging part was to allow a synchronization of over 600,000 files and folders to the mobile device. A service had to be provided to retrieve the files since they are stored on file servers inside the company's IT infrastructure.

The company also had specific ideas how the application should look like. Besides the common adjustments of the user interface to match the corporate identity of the company they wanted to change the appearance of files and folders in the user interface when using the grid mode of the browsing feature. Therefore a new cell design should be implemented.

The folder structure this department of the company is using is managed in a very depth way. Hence the user has to enter a big number of folders and subfolders to get to a specific file. The company asked for a shortcut feature to allow quick jumps to user defined folders. With this feature the user of the application can trigger the shortcut to access a folder without entering all the parent folders. Another requirement was to include favorites. These favorites should be downloaded automatically and therefore these marked files should always be up to date and available offline.

The department wanted to decide about the ordering of files and folders within the application. They renamed the items in their file systems by appending a prefix. This prefix was used to bypass the alphabetical ordering of the file system. They wanted the application to use this prefix to order the results in the user interface but should display the name of the folders without the defined prefix. The prefixes are defined as two digit numbers followed by a whitespace and the name of the file or folder. Table 7.3 sums up the requirements of the customer and maps them to features of the feature model.

Requirement	Feature
Custom Sync Engine	Sync Engine - Sync Type
Custom Sorting of Files	Sync Engine
Custom Download Type (replicated Favorites)	Sync Engine - Download Type
Custom Cell for Folders / files	Custom Component
User Interface Customization	Custom Layout - Stylesheet
Shortcuts to Folders	File Browsing - Shortcuts
Favorites	File Management - Favorites

Table 7.3: Overview of requirements and features for the anonymous furniture department store.

### 7.3.2 Customization Details

The steps for creating a new product were followed to create a start point for the product derivation. After that, a sync service had to be implemented to connect to the file servers of the company's IT infrastructure. Due to the amount of files a decision was made to use a delta sync engine to sync all the files to the mobile device. One big problem while developing the sync engine was the size of the server response during the initial sync process. The response JSON was up to 50 MB, hence it was not possible to use the standard built-in way to load a JSON because of memory issues of the mobile device. A SAX style parser was used to minimize the memory usage while parsing the JSON file. To reduce the time spent for the initial sync a method was implemented which allows parsing of a JSON file while the file is downloading. When using a fast connection to the file server the application will not be able to persist the parsed file fast enough. Hence the parsed data will be kept in memory again which also leads to memory issues. Therefore this enhancement got disabled again. The application is now downloading the response and saving it to the file system of the mobile device and will start with the parsing process after the download. After successfully persisting all the data into the database the file will be deleted from the file system.

The data model of the application got extended by the file's property `nameForSorting`. The application will use this property in every screen to sort the files and folders but will display the actual name of the file which is stored inside the property `name`. The sync service got adopted to save the full name of the file including the prefix in this new property while saving the name without the prefix in the property `name`.

The module for favorites had to be included into the product. Therefore the *Modules.plist* files which is represented as array got extended by the entry `CPNFavoriteModule` which is the class name of the subclass of `CPNBaseModule`. Favorites and custom collections are designed as tags with a specific name, namely *Favorites* for favorites and a user chosen name for a custom collection. These tags have a property in the data model which is called `availableOffline`. If set to `true` all files with this tag are downloaded automatically (replication mode). The property of the favorites tag has the default value `true`, hence favorites will get downloaded automatically.

To enable the action to create shortcuts to folders the property `folderActions` of the class



`CPNGlobalConfiguration` was extended with this file action.

All requested user interface adjustment were handled with a stylesheet for the anonymous furniture department store. The custom cell was not manageable with setting properties with the stylesheet and therefore a new class has to be implemented. The class was implemented by subclassing `CPNCollectionViewCell` which is used as default cell representation in the application. This new subclass `XMLCollectionViewCell` adjusts the existing cell to fulfill the required design changes. To load this substitute cell class the *Class Loader* configuration file was adjusted by adding an entry with the key `CPCollectionViewCell` and `XMLCollectionViewCell` as value. A summary of the required steps and the resulting effort is given in Table 7.4.

Category	Description / Metric
Changes in Code Base	Extended data model with property <code>nameForSorting</code> to allow custom sorting of files and folders with a separate property
Number of Configuration Files	3
Number of overridden Classes	3
Number of new Classes	2
Number of additional Resources	2
Number of Lines for Stylesheet	395

Table 7.4: Overview of effort for product for the anonymous furniture department store.

## 7.4 Mercedes Benz Austria - MBÖ App2Date

### 7.4.1 Domain and Requirement Description

The marketing department of Mercedes Benz Austria<sup>2</sup> decided to improve their selling process to communicate a young spirit and a technological progress by using an iPad application. This iPad application is used during the customer pitch to show videos, images and catalogues of their cars as well as technical specifications or detailed price information.

For each customer a collection of files which the customer is interested in can be created and sent to him after the customer pitch.

An iPad has been sent to every Mercedes salesperson in Austria as well as a download link to the application. The fact that every salesperson has this application installed on the application opens Mercedes a new way to communicate with their employees which are spread over Austria. Therefore they wanted Apple's *Push Notification* included into the application to inform their employees about important news.

---

<sup>2</sup><http://www.mercedes-benz.at>

Since every salesperson will have an iPad with this application they wanted to use this application not only to share product information but also business relevant information like memos and other sales related documents. These documents were sent as a letter to all the salespersons.

Due to the fact that the application contains sensitive data containing information about the sales process it is very important to Mercedes that only people are using this application which are allowed to. Therefore a person who is getting an iPad with the application should first of all register within the application with its name, contact information and location. After this registration step the support team will be checking with the marketing department if the user is eligible to use this application and if so decide which files the user is allowed to view. After enabling the users right for the salesperson the application should continue the start up process and sync the files.

The users of the application may not have a reliable internet connection during field work. To prevent that a salesperson is not able to show the product information to a future customer all files should be downloaded automatically during sync.

Mercedes Benz requested a *Private Mode* to prevent the salesperson from showing sensitive files to customers like the above mentioned memos. The user should be able to switch to the private mode by entering an assigned PIN. After successfully entering the pin the user should be able to see all files, sensitive and nonsensitive ones. When leaving and reopening the application the application should be in public mode again. The user interface should reflect the current mode, by applying a blue color to the status bar at the bottom of the user interface elements.

Another customization of the user interface besides the adjustments to match corporate identity was requested by renaming the root element of the breadcrumb bar. The breadcrumb bar gets used to quickly navigate to parent folders. The first button of this breadcrumbs is named *All Files* per default. Mercedes Benz requested a renaming of this button as additional requirement. Table 7.5 sums up the requirements of the customer and maps them to features of the feature model.

Requirement	Feature
Custom Sync Engine	Sync Engine - Sync Type
Replication	Sync Engine - Download Type
Customer Catalogue	File Management - Custom Collection
Custom Registration Dialog	Custom Component
Private Mode	Custom Component / Additional Behavior and Functionality
Custom Status Bar	Custom Component
User Interface Customization	Custom Layout - Stylesheet

Table 7.5: Overview of requirements and features for Mercedes Benz Austria.

### 7.4.2 Customization Details

Mercedes Benz is not accessing these files in an existing file system and therefore asked for a solution to host the files on a new server. Hence it was possible to provide the same server setup as for AESchreder which allowed to use the same sync service. The sync engine for AESchreder therefore got moved from the submodule for this product to the code base of the software product line, it therefore became one of the core assets. The different server URL was configured using a configuration file and the *Configuration Loader* mechanism.

The *Custom Collection* was used to allow the grouping of files for a customer into a collection. This module was included into the application by extending the *Module.plist* with the entry *CPNWorkbookModule* which is the class name for this module. The module provider therefore will load this module into the menu on every application start.

To display a registration dialog new classes and views were created. To override the startup procedure of the application a subclass of the splash screen was created. This subclass verifies if the device is already registered and if the device has any groups associated. When no such registration was done before a dialog is collecting data to identify the user and will create a user token which is saved on the device. If a token can be found on the device the user already has registered on the server. As long as this token has access groups associated the application will continue launching and will be ready to use. This token which identifies the user will be used after the registration process to automatically authenticate the device with the server. To specify this splash screen as substitute of the existing one the configuration file of the *Class Loader* got extended with the entry with the key `CPNSplashViewController` and the value `MBOSplashViewController`.

To allow the user to switch between the public and private mode the `CPNMenuViewController`<sup>3</sup> got subclassed. This new subclass `MBOMenuViewController` was extended with a switch to enter or leave the private mode. When enabling this switch a keypad will be presented to verify the user's PIN. Another view was created to represent this keypad. The substitute for the menu screen was registered in the configuration file of the *Class Loader*.

The status bar also got overridden, to react on a change of the mode (from public to private or vice versa). When enabling the private mode, the status bar will color itself in a blue color, when switching back to the public mode, the color will be adjusted to black. The substitute for the status bar also got registered for the *Class Loader*.

As an additional change to the code base the text of the first button in the breadcrumb bar was exposed as additional property. Therefore it is possible to set this property to a custom value by using the *Configuration Loader*. All other required design changes were made by creating a stylesheet for the application. A summary of the required steps and the resulting effort is given in Table 7.6.

---

<sup>3</sup>This class is responsible for displaying all modules in the left drawer as well as displaying a settings area for the application.

Category	Description / Metric
Changes in Code Base	Moved sync engine to code base, since it was used from several products. Exposed title of first button of breadcrumb bar.
Number of Configuration Files	3
Number of overridden Classes	9
Number of new Classes	3
Number of additional Resources	6
Number of Lines for Stylesheet	509

Table 7.6: Overview of effort for product for Mercedes Benz Austria.

## 7.5 Critical Discussion

The evaluation brought up, that customers do have a lot of different requirements that have to be implemented with custom components and heavy customization. Therefore a way had to be found to easily exchange whole parts of the application with a substitute or extend the application with new features that might not be reused from any other customer. The introduced *Module Concept* is handling the problem with an extensible interface very well, but a lot of requirements force changes on a lower abstraction level than a module is offering. In such cases the *Class Loader* concept was the right choice to handle these kind of requirements. It is acting like a tool to extend the base version of the software product line with the customer specific needs.

This derivation of the original *Subclassing* variability mechanism does not seem as elegant as an aspect-oriented approach a lot of scientific papers are analyzing. Since the lack of an aspect-oriented approaches of Objective-C and the complexity and dependencies that might be introduced by using such an approach, the chosen variability mechanisms are good alternatives which are easy to use and understand.

While creating the different products for the evaluation, the time which was necessary to build the products was a very interesting metric to observe. One negative impact on that factor, was the presence of different sync engines. Although the class concept behind the sync engine was very convincing, various `SyncClients` and `SyncServices` had to be implemented for the different customers, due to the different sync services they are using. Nevertheless the concept of `SyncEngines`, which are acting with sync services that have to conform to specific protocols, was accurate.

The variation point, that was the most time-consuming one, was definitely the style sheet feature (see Table 7.7). Beside the time, that had to be spent to create the timesheet, the chosen implementation with *UISS* implies an additional effort in training to be able to use this framework efficiently. The ability to define variables for colors and other properties to reuse them in the stylesheet, has to be used, to make a modification of stylesheets more flexible. Another way of creating the stylesheet should be analyzed, to decrease the time to market.

When comparing the three stylesheets generated for the scenarios it can be observed that

a lot of parts do not differ between them. The structure of the *JSON* file is nearly the same, almost only the values of the entries of that dictionary differ. When heavily using variables to define colors, shadows and background patterns a stylesheet template can be created to fasten up the process of stylesheet creation. The creation of the stylesheet could also be handled by generation. This generation could use definitions like colors as inputs and produce a generated stylesheet as output. This output could be modified if necessary.

Generation in general is also mentioned a lot in scientific researches, unfortunately no tools for such code generation exist for Objective-C and Cocoa Touch.

As mentioned in this thesis, configuration management is very important for software product lines to track the different versions of core assets among the built products. The solution developed during this thesis with the product related code and configuration in a separate *GIT* submodule was working well during the evaluation. The build number is marking the version of the code base and therefore also the version of the core assets. The applications version number is marking the version number of one single product and therefore is persisted inside the separate submodule. The combination of product identifier (that is the bundle identifier of the iOS application), the build number and the version number are marking releases of one specific product. The fact, that for each release a tag is created, makes it easy to track all changes and to rollback if necessary. In spite of this, tagging and the steps for creating a new product can be simplified or partly automatized to decrease possible failures.

Category	Metric
Total Number of Changes in Code Base	5
Total Number of Configuration Files	10
Total Number of overridden Classes	14
Total Number of new Classes	6
Total Number of additional Resources	10
Average Number of Lines for Stylesheet	447

Table 7.7: Total effort for the three derived products.



## Conclusion and Future Work

Nowadays, data storage is shifted to the cloud. At the same time, mobile applications are used within companies to improve their business processes. However, the mobile applications provided by the cloud-storage providers do not satisfy the needs of these companies because of reasons such as mentioned in Chapter 1.

Therefore, in this thesis, requirements were analyzed from the domain of cloud storage-based mobile applications. These requirements were collected from customer pitches and in form of informal interviews with project managers. The collected requirements were grouped into topics covering file synchronization, file browsing, meta information synchronization and manipulation, file management and file organization, communication aspects, customizable user interface and other custom modifications.

Afterwards, principles of software product lines were discussed in detail. The processes of software product line engineering was examined and existing modeling techniques to model variability were introduced. The variability mechanisms to implement variability in a software product line were listed and explained. Other papers in the area of software product lines with a mobile context were mentioned to distinguish this thesis from others. It is shown that no papers exist which are discussing variability in combination with iOS or Objective-C. The papers found which discuss software product lines in a mobile context are covering older technologies like J2ME. An analysis including newer techniques which are related to multi-touch devices could not be found.

This thesis also introduced the basic concepts of Objective-C to create a knowledge base for the remaining chapters. This knowledge base was used to show how variability mechanisms can be implemented with Objective-C. These implementations were combined with object-oriented programming techniques and design patterns to create tools to introduce variability to a Objective-C-based mobile software product line.

The established requirements were used to derive features for the application. With these features a feature model for a software product line was constructed. After this, the features were mapped to the implementation techniques developed before. This mapping showed how variability can be introduced efficiently in mobile applications. This software product line was

evaluated by using the descriptive evaluation method [HMPR04]. In particular, three scenarios were chosen to derive products from the software product line. These products have satisfied the customer's requirements, therefore, this evaluation underlines the applicability of the meet design decisions.

In this chapter, conclusions are drawn about software product line engineering in a mobile context and an outline is drawn about future work.

## 8.1 General Observations

The decision to plan and implement a software product line should be well-considered. The example given in this thesis shows, that an implementation of such a product line is more complex than for standard software products, because variation points have to be kept in mind, the choice of variability mechanism has to be well-chosen and the architecture has to be more flexible, extensible and maintainable.

Another key aspect is represented by the selection of the **scope** and the **features** of the product line. When making the wrong decisions, core assets might be used only in a couple of products (in case there are too many of them), or a lot of core assets have to be developed while implementing new products (in case there are too little of them).

An observation while writing this thesis, talking to customers and selling the product line was made, that a well-chosen feature model is also acting as **marketing** tool. Customers rethink their requirements and tend to add features, that they haven't planned to integrate in their application before. It also seems, that they adjust their needs to match the predefined solutions if they differ not too much.

Another benefit can be gained of new features customers are requesting. As part of the **feedback loop** of the application engineering, some of them might be very interesting to integrate as core assets into the product line. Hence the much more fine-grained file action requirement from AESchreder was integrated as core asset and indeed Mercedes was also requesting a similar control of allowed actions on files and folders.

## 8.2 Challenge of Mobile Applications

A lot of challenges were brought up while developing the feature model of the application and also while implementing the different variability mechanisms. Besides the features of an application, the **user interface and interaction** is the most important part of a mobile application. Most of the features of such an application are present inside the user interface, therefore variability which comes with these features also influences the applications look. Creating a concept, how the user interface should react on extensibility and changes through the introduced variability was one of the most challenging parts.

Not only the variability introduced passively into the user interface was challenging, but also the actively introduced, namely the ability to apply different **styles** to the interface. Customers, which want to invest in an enterprise application want to have their corporate identity represented in the application. They do not want a standard interface, but want to have their colors, logos, fonts, etc. present in the application, the application should look like a product of theirs.



## 8.3 Future Work

As mentioned in Chapter 7, the stylesheet creation is time-consuming and error-prone, therefore a method to generate the stylesheet by defining a set of colors will be provided. This might be extended by an application, which acts like a wizard. This wizard should guide the developer through different steps, where the developer can choose colors and gradient types for different areas of the application. At the end of the wizard a stylesheet should be generated, which can be modified to get a more fine-grained result.

A what-you-see-is-what-you-get (WYSIWYG) editor is desirable, but would lead to a huge amount of cost, which will only be amortized after a large number of products built from the software product line. Therefore such an editor will not be economically efficient, but very conformable for the developer or designer.

The development of new core assets and reevaluating of existing ones after building new products should always be kept in mind to improve the quality of the software product line.

The benefits, which may come with porting this software product line to the iPhone will be evaluated. A *Web(comp)anion*, a web version of the software product line, was also discussed and may be a step to make in the future.



# Bibliography

- [AC04] Michal Antkiewicz and Krzysztof Czarnecki. FeaturePlugin: feature modeling plug-in for Eclipse. In *Proceedings of the 2004 OOPSLA workshop on Eclipse Technology eXchange, ETX 2004, Vancouver, British Columbia, Canada, October 24*, pages 67–72, 2004.
- [AMJC<sup>+</sup>05] Vander Alves, Pedro Matos Jr, Leonardo Cole, Paulo Borba, and Geber Ramalho. Extracting and evolving Mobile Games product Lines. In *Software Product Lines*, pages 70–81. Springer, 2005.
- [App13a] Apple Inc. *App Distribution Guide*, 2013. <http://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/Notifications/Notifications.pdf>.
- [App13b] Apple Inc. *Core Data Programming Guide*, September 2013. <https://developer.apple.com/library/ios/documentation/cocoa/conceptual/CoreData/CoreData.pdf>.
- [App13c] Apple Inc. *Key-Value Coding Programming Guide*, 2013. <https://developer.apple.com/library/mac/documentation/cocoa/conceptual/KeyValueCoding/KeyValueCoding.pdf>.
- [App13d] Apple Inc. *Notification Programming Topics*, 2013. <http://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/AppDistributionGuide.pdf>.
- [App13e] Apple Inc. *Objective-C Runtime Programming Guide*, 2013. <http://developer.apple.com/mac/library/documentation/cocoa/conceptual/ObjCRuntimeGuide/ObjCRuntimeGuide.pdf>.
- [App13f] Apple Inc. *Programming with Objective-C*, 2013. <http://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/ProgrammingWithObjectiveC.pdf>.
- [Bas96] Paul G Bassett. *Framing Software Reuse: Lessons From the Real World*. Prentice-Hall, Inc., 1996.

- [BFG00] Joachim Bayer, Oliver Flege, and Cristina Gacek. Creating Product Line Architectures. In *Proceedings of Software Architectures for Product Families, International Workshop IW-SAPF-3, Las Palmas de Gran Canaria, Spain, March 15-17*, pages 210–216. 2000.
- [BFG<sup>+</sup>01] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J. Henk Obbink, and Klaus Pohl. Variability Issues in Software Product Lines. In *Software Product-Family Engineering, 4th International Workshop, PFE 2001, Bilbao, Spain, October 3-5, Revised Papers*, pages 13–21. 2001.
- [BKPS04] G Böckle, P Knauber, K Pohl, and K Schmid. *Software-Produktlinien: Methoden, Einführung und Praxis*. dpunkt.verlag, 2004.
- [CGR<sup>+</sup>12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. Cool features and tough decisions: a comparison of variability modeling approaches. In *Proceedings of sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27*, pages 173–182, 2012.
- [CHE04] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged Configuration Using Feature Models. In *Proceedings of Software Product Lines, Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2*, pages 266–283. 2004.
- [CJ01] Matthias Clauß and Intershop Jena. Modeling variability with uml. In *GCSE 2001 Young Researchers Workshop*. Citeseer, 2001.
- [CK05] Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-based feature modeling and constraints: A progress report. In *Proceedings of the 2005 OOPSLA Workshop on Software Factories, San Diego, California, USA, October 17, 2005*, 2005.
- [Cla01a] Matthias Clauß. A Proposal for Uniform Abstract Modeling of Feature Interactions in UML. In *Proceedings of the ECOOP 2001 Workshop on Feature Interaction in Composed Systems (FICS 2001), Budapest, Hungary, June 18-22, 2001*, pages 21–25, 2001.
- [Cla01b] Matthias Clauß. Generic Modeling using UML Extensions for Variability. In *Workshop on Domain Specific Visual Languages at OOPSLA*, volume 2001, 2001.
- [Cla01c] Matthias Clauß. Untersuchung der Modellierung von Variabilität in UML, 2001.
- [CN02] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
- [Cor93] Software Productivity Consortium Services Corporation. *Reuse-driven Software Processes Guidebook: SPC-92019-CMC, Version 02.00. 03*. Software Productivity Consortium Services Corporation, 1993.

- [CP06] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proceedings of Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26*, pages 211–220, 2006.
- [DGR07] Deepak Dhungana, Paul Grünbacher, and Rick Rabiser. DecisionKing: A Flexible and Extensible Tool for Integrated Variability Modeling. In *Proceedings of First International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS 2007, Limerick, Ireland, January 16-18*, pages 119–127, 2007.
- [Eic12] Michael Eichberg. Enterprise application design. Technische Universität Darmstadt, Department of Computer Science, Germany, 2012.
- [GA01] Cristina Gacek and Michalis Anastasopoulos. Implementing Product Line Variabilities. In *Proceedings of the 2001 Symposium on Software Reusability (SSR '01): Putting Software Reuse in Context*, pages 109–117, 2001.
- [GHJ<sup>+</sup>95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, et al. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Boston; MA, 1995.
- [HMPR04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–105, 2004.
- [IKPJ11] Paul Istóan, Jacques Klein, Gilles Perouin, and Jean-Marc Jézéquel. A Metamodel-based Classification of Variability Modeling Approaches. In *Proceedings of VARY International Workshop affiliated with ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, pages 23–32, 2011.
- [Jaa02] Ari Jaaksi. Developing Mobile Browsers in a Product Line. *IEEE Software*, 19(4):73–80, 2002.
- [JGJ97] Ivar Jacobson, Martin L. Griss, and Patrik Jonsson. *Software Reuse - Architecture, Process and Organization for Business*. Addison-Wesley-Longman, 1997.
- [KCH<sup>+</sup>90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University Software Engineering Institute, Pittsburgh, PA, USA, 1990.
- [KKL<sup>+</sup>98] Kyo Chul Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euseob Shin, and Moonhang Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Software Eng.*, 5:143–168, 1998.

- [KL07] Birgit Korherr and Beate List. A UML 2 Profile for Variability Models and their Dependency to Business Processes. In *Proceedings of 18th International Workshop on Database and Expert Systems Applications (DEXA 2007)*, 3-7 September, Regensburg, Germany, pages 829–834, 2007.
- [KLM<sup>+</sup>97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. *ECOOP'97—Object-Oriented Programming*, pages 220–242, 1997.
- [MHP<sup>+</sup>07] Andreas Metzger, Patrick Heymans, Klaus Pohl, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. In *RE*, pages 243–253, 2007.
- [Nor02] Linda M. Northrop. Sei's software product line tenets. *IEEE Software*, 19(4):32–40, 2002.
- [Obj13] Object Management Group. *Object Constraint Language (OCL) – ISO/IEC 19507*, 2013. <http://www.omg.org/spec/OCL/ISO/19507/PDF>.
- [PBvdL05] Klaus Pohl, Günter Böckle, and Frank J van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
- [Pou97] Jeffrey S. Poulin. Measuring software reuse - principles, practices, and economic models. pages I–XIX, 1–195, 1997.
- [Rei97] Donald J Reifer. *Practical Software Reuse*. John Wiley & Sons, Inc., 1997.
- [Sch95] Douglas C. Schmidt. Using Design Patterns to Develop Reusable Object-Oriented Communication Software. *Commun. ACM*, 38(10):65–74, 1995.
- [SRG11] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. A Comparison of Decision Modeling Approaches in Product Lines. In *Proceedings of Fifth International Workshop on Variability Modelling of Software-Intensive Systems, Namur, Belgium, January 27-29*, pages 119–126, 2011.
- [SvGB05] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A Taxonomy of Variability Realization Techniques. *Softw., Pract. Exper.*, 35(8):705–754, 2005.
- [Tra95] Will Tracz. *Confessions of a Used-Program Salesman: Lessons Learned*. 1995.
- [WL99] David M Weiss and Chi Tau Robert Lai. *Software Product Line Engineering: A Family based Software Engineering Process*. Addison-Wesley, 1999.
- [Wor13] World Wide Web Consortium. *XML Path Language (XPath) 2.0*, 2013. <http://www.w3.org/TR/xpath20/>.
- [You05] Trevor J Young. *Using AspectJ to build a Software Product Line for Mobile Devices*. PhD thesis, The University of British Columbia, 2005.

- [Zar13] Marcus S. Zarra. *Core Data: Data Storage and Management for iOS, OS X, and iCloud*, volume 2. Pragmatic Bookshelf, 2013.