

A TEST ORIENTED SERVICE AND OBJECT MODEL
FOR SOFTWARE PRODUCT LINES

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

NAZİF BÜLENT PARLAKOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

APRIL 2010

Approval of the thesis

**A TEST ORIENTED SERVICE AND OBJECT MODEL
FOR SOFTWARE PRODUCT LINES**

submitted by **NAZİF BÜLENT PARLAKOL** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering**

Asst. Prof. Dr. Pınar Şenku
Supervisor, **Computer Engineering Dept., METU**

Examining Committee Members:

Assoc. Prof. Dr. Ali H. Doğru
Computer Engineering Dept., METU

Asst. Prof. Dr. Pınar Şenku
Computer Engineering Dept., METU

Asst. Prof. Dr. Tolga Can
Computer Engineering Dept., METU

Esen Kaçar, M.Sc.
MVS Spatial Data Systems

Ahmet Serkan Karataş, M.Sc.
K & K Technology

Date:

29.04.2010

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Nazif Bülent Parlakol

Signature :

ABSTRACT

A TEST ORIENTED SERVICE AND OBJECT MODEL FOR SOFTWARE PRODUCT LINES

Parlakol, Nazif Bülent

M.Sc., Department of Computer Engineering

Supervisor: Asst. Prof. Dr. Pınar Şenkul

April 2010, 99 pages

In this thesis, a new modeling technique is proposed for minimizing regression testing effort in software product lines. The “Product Flow Model” is used for the common representation of products in application engineering and the “Domain Service and Object Model” represents the variant based relations between products and core assets. This new approach provides a solution for avoiding unnecessary work load of regression testing using the principles of sub-service decomposition and variant based product/sub-service traceability matrices. The proposed model is adapted to a sample product line targeting the banking domain, called Loyalty and Campaign Management System, where loyalty campaigns for credit cards are the products derived from core assets. Reduced regression test scope after the realization of new requirements is demonstrated through a case study. Finally, efficiency improvement in terms of time and effort in the test process with the adaptation of the proposed model is discussed.

Keywords: Software Product Line, Product Flow Model, Service and Object Model, Test Oriented Modeling, Regression Testing

ÖZ

YAZILIM ÜRETİM BANTLARI İÇİN TESTE YÖNELİK HİZMET VE NESNE MODELİ

Parlakol, Nazif Bülent

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Yrd. Doç. Dr. Pınar Şenkul

Nisan 2010, 99 sayfa

Bu tezde, yazılım üretim bantlarında regresyon test maliyetlerinin en aza indirgenmesi için yeni bir modelleme tekniği önerilmiştir. “Ürün Akış Modeli” uygulama mühendisliğindeki ürünlerin ortak gösterimi için kullanılmakta ve “Alan Servis ve Nesne Modeli” ürünler ve merkez varlıklar arasındaki değişkenlere bağlı ilişkileri göstermektedir. Bu yeni yaklaşım alt-servis ayrıştırması prensiplerini ve değişkenlere bağlı ürün/merkez varlık izlenebilirlik matrislerini kullanarak gereksiz regresyon test yükünden kaçınmak için bir çözüm sağlamaktadır. Önerilen model, merkez varlıklardan kredi kartı sadakat kampanyalarının ürün olarak türetildiği, bankacılık alanını hedef alan örnek bir yazılım üretim bandı olan Sadakat ve Kampanya Yönetim Sistemi’ne uyarlanmıştır. Yeni gereksinimlerin gerçekleşmesi sonrasında indirgenmiş regresyon test kapsamı örnek olay incelemesi yoluyla gösterilmiştir. Sonuç olarak, önerilen modelin uyarlanması ile test sürecindeki zaman ve maliyet anlamında verimlilik artışı ele alınmıştır.

Anahtar Kelimeler: Yazılım Üretim Bandı, Ürün Akış Modeli, Hizmet ve Nesne Modeli, Teste Yönelik Modelleme, Regresyon Testi

To the future

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude and appreciation to Assoc. Prof. Dr. Ali H. Doğru for his valuable guidance, support and concern on this thesis.

I would like to thank my supervisor Asst. Prof. Dr. Pınar Şenkul, and I want to express sincere appreciation to Esen Kaçar and Asst. Prof. Dr. Tolga Can for being in the examining committee and all their valuable comments.

I would like to convey special thanks to A. Serkan Karataş for his great support and encouragement. He was again with me as he has always been before.

I would like to thank all my colleagues working with me for the LCMS project, for their effort in the success of the project, and especially to Rana Toprak, the most sophisticated and the smartest analyst I have ever worked with, for her miscellaneous impact and remarkable support on the completion of this thesis.

I am deeply grateful to my wife Fatma Parlakol for her endless love and motivating support during this study.

And finally, I owe my little daughter Sila Parlakol an apology for spending considerable time working on this thesis instead of playing with her.

TABLE OF CONTENTS

ABSTRACT	iv
ÖZ	v
DEDICATION	vi
ACKNOWLEDGEMENTS	vii
TABLE OF CONTENTS	viii
LIST OF TABLES	xii
LIST OF FIGURES	xv
LIST OF ABBREVIATIONS	xvii
CHAPTER	
1. INTRODUCTION	1
2. SOFTWARE PRODUCT LINES IN GENERAL	3
2.1. What is a Software Product Line	3
2.2. Software Product Line Processes	4
2.2.1. Domain Engineering	5
2.2.2. Application Engineering	6
2.3. Software Product Line Management	7
2.4. Software Product Line Practice Areas	8
2.5. Transition to a Software Product Line	11
2.5.1. Transition Strategies and Approaches	11
2.5.2. Organization Aspects	13
2.5.2.1. Hierarchical Organization Structures	14
2.5.2.2. Orientation-Based Organization Structures	17
2.5.2.3. Matrix Organization Structures	18

2.5.3. Software Product Line Maturity	19
2.6. Benefits of Software Product Line Engineering	22
3. VARIABILITY IN SOFTWARE PRODUCT LINES	25
3.1. Principles of Variability	25
3.2. Binding Times	27
3.3. Variability in Time and Space	28
3.4. Levels of Variability	29
3.5. Implementing Variability	30
3.6. Representation of Variability	31
3.6.1. Feature Models	33
3.6.2. Orthogonal Variability Model	33
4. TESTING A SOFTWARE PRODUCT LINE	35
4.1. The Testing Context	36
4.2. Software Product Line Testing vs. Single System Testing	38
4.3. Creating Reusable Test Artifacts	39
4.4. Domain Testing (Testing Core Assets)	40
4.5. Application Testing (Testing Products)	43
5. CURRENT SYSTEM SPECIFICATION	45
5.1. Loyalty and Campaign Management System	45
5.2. Evolution of LCMS as a Software Product Line	47
5.2.1. Organizational Structure and Processes	47
5.2.2. Domain Engineering for LCMS	48
5.2.3. Application Engineering for LCMS	49
5.2.4. Commonality and Variability in LCMS	51
5.3. Problem Definition: Overhead in Regression Testing	53
6. TEST ORIENTED SERVICE AND OBJECT MODEL	54
6.1. Application Engineering – Product Flow Model	54
6.2. Domain Engineering – Service and Object Model	56

6.2.1. Specifications for Domain Object Modeling	56
6.2.2. Specifications for Domain Service Modeling	57
6.3. Test Orientation of the Model	61
7. CASE STUDY: MODELING LCMS	62
7.1. Scope	62
7.2. Implementation of the Proposed Model on LCMS	62
7.2.1. Product Flow Model for LCMS	63
7.2.2. Modeling Domain Objects of LCMS	65
7.2.3. Modeling Domain Services of LCMS	65
7.3. Defining Test Scope after Extensions to LCMS	68
7.3.1. Latest Requirements	69
7.3.2. Building a New Application Product	69
7.3.3. Extending Core Assets	69
7.3.3.1. Changes in Domain Objects	70
7.3.3.2. Changes in Domain Services	70
7.3.4. Test Scope after Extensions	71
8. CONCLUSIONS AND FUTURE WORK	73
8.1. Conclusions	73
8.2. Future Work	76
REFERENCES	77
APPENDICES	80
A. DOMAIN OBJECT SPECIFICATIONS OF LCMS	80
A.1. Target Lists	80
A.2. Campaign Counters	81
A.3. Ledger Records	81
A.4. Transaction Logs	82
A.5. Reports	83
B. DOMAIN SERVICE SPECIFICATIONS OF LCMS	84

B.1. Campaign Entrance Control	84
B.2. Target Group Decision	88
B.3. Campaign Counter Update	91
B.4. Ledger Record Creation	95
B.5. Transaction Log Creation	98

LIST OF TABLES

TABLES

Table 2.1	Software product line practice areas	9
Table 2.2	Compatibility relation of maturity levels to organizational structures	21
Table 2.3	Compatibility relation of maturity levels to artifacts	22
Table 3.1	Variability Mechanisms	30
Table 3.2	Feature Relation Types	33
Table 4.1	Static testing techniques for non-software core assets	42
Table 5.1	Application size of LCMS	46
Table 5.2	Usage of LCMS products on distribution channels	51
Table 5.3	Examples of external variability in LCMS	52
Table 5.4	Examples of internal variability in LCMS	52
Table 6.1	Document template for domain object specification	57
Table 6.2	Document template for domain service description	58
Table 6.3	Document template for sub-service decomposition	58
Table 6.4	Matrix template for sub-service dependencies on variants	59
Table 6.5	Matrix template for product bindings on variants	59
Table 6.6	Matrix template for variant based product/sub-service traceability	60
Table 7.1	Domain object specifications for Reward Pools	65
Table 7.2	Domain service description for Reward Pool Update	66
Table 7.3	Sub-service decomposition for Reward Pool Update	67
Table 7.4	Sub-service dependencies for Reward Pool Update	67
Table 7.5	Product bindings for Reward Pool Update	67
Table 7.6	Traceability matrix for Reward Pool Update	68

Table 8.1	Efficiency throughput for the test oriented service and object model	74
Table A.1	Domain object specifications for Target Lists	80
Table A.2	Domain object specifications for Campaign Counters	81
Table A.3	Domain object specifications for Ledger Records	81
Table A.4	Domain object specifications for Transaction Logs	82
Table A.5	Domain object specifications for Reports	83
Table B.1	Domain service description for Campaign Entrance Control	84
Table B.2	Sub-service decomposition for Campaign Entrance Control	85
Table B.3	Sub-service dependencies for Campaign Entrance Control	86
Table B.4	Product bindings for Campaign Entrance Control	86
Table B.5	Traceability matrix for Campaign Entrance Control	87
Table B.6	Domain service description for Target Group Decision	88
Table B.7	Sub-service decomposition for Target Group Decision	89
Table B.8	Sub-service dependencies for Target Group Decision	89
Table B.9	Product bindings for Target Group Decision	90
Table B.10	Traceability matrix for Target Group Decision	90
Table B.11	Domain service description for Campaign Counter Update	91
Table B.12	Sub-service decomposition for Campaign Counter Update	92
Table B.13	Sub-service dependencies for Campaign Counter Update	92
Table B.14	Product bindings for Campaign Counter Update	93
Table B.15	Traceability matrix for Campaign Counter Update	94
Table B.16	Domain service description for Ledger Record Creation	95
Table B.17	Sub-service decomposition for Ledger Record Creation	96
Table B.18	Sub-service dependencies for Ledger Record Creation	96
Table B.19	Product bindings for Ledger Record Creation	96
Table B.20	Traceability matrix for Ledger Record Creation	97
Table B.21	Domain service description for Transaction Log Creation	98
Table B.22	Sub-service decomposition for Transaction Log Creation	99

Table B.23	Sub-service dependencies for Transaction Log Creation	99
Table B.24	Product bindings for Transaction Log Creation	99
Table B.25	Traceability matrix for Transaction Log Creation	99

LIST OF FIGURES

FIGURES

Figure 2.1	Overview of basics in a software product line [5]	4
Figure 2.2	The software product line engineering framework [2]	5
Figure 2.3	The three essential software product line activities [1]	8
Figure 2.4	Software product line practice areas [5]	10
Figure 2.5	Relations among practice areas of a software product line [5]	10
Figure 2.6	Development department model [9]	15
Figure 2.7	Business units model [9]	15
Figure 2.8	Domain engineering unit model [9]	16
Figure 2.9	Hierarchical domain engineering units model [9]	16
Figure 2.10	Product-oriented organization [3]	17
Figure 2.11	Process-oriented organization [3]	18
Figure 2.12	Sample matrix organization with testing as a separate functional unit	19
Figure 2.13	Maturity levels for software product lines [10]	20
Figure 2.14	Reduction in development costs in a software product line [6]	23
Figure 2.15	Enhancement of quality due to reduction in the number of defects [6]	23
Figure 2.16	Reduction in time to market [2]	24
Figure 3.1	Partially instantiated binding times through development stages [6]	27
Figure 3.2	Variability in time for domain and application engineering [15]	28
Figure 3.3	Levels of abstraction for variability [17]	29
Figure 3.4	Graphical notation for orthogonal variability model [3]	34
Figure 4.1	Test and development processes [25]	37
Figure 4.2	Test implementation in product line engineering [22]	37

Figure 4.3	Information flow on application testing [2]	43
Figure 5.1	Distribution channels integrated to LCMS	46
Figure 5.2	Sample channel integration of LCMS	47
Figure 6.1	Graphical notation for product flow model	55
Figure 7.1	Product flow model for LCMS	64
Figure 8.1	Efficiency graph for the test oriented service and object model	75

LIST OF ABBREVIATIONS

LCMS	-	Loyalty and Campaign Management System
POS	-	Point Of Sale
RFM	-	Recency, Frequency, Monetary Value
CCB	-	Common Cash Back
PCB	-	Private Cash Back
XCB	-	Cross Cash Back
CIF	-	Customer Information File
CMS	-	Card Management System
KLOC	-	Kilo Lines of Code
JCL	-	Job Control Language

CHAPTER 1

INTRODUCTION

A new approach to software reuse, which is known as software product line development, has gained considerable attention both by industry and academia over the past few years. Studies have shown that organizations can yield remarkable improvements in productivity, time to market, product quality and customer satisfaction by applying this approach. The characteristic that distinguishes software product lines from previous efforts is predictive versus opportunistic software reuse. The basic concept of software product line engineering is the separation of the process as *Domain Engineering* including the core assets and *Application Engineering* constructing specific products by the effective reuse of core assets.

Software product line engineering aims at supporting a range of products which may serve different customers or market segments. Instead of understanding each individual system by itself, software product line engineering looks at the product line as a whole and concentrates on the variation among the individual systems. The variability and commonality of products must be managed throughout software product line engineering.

Testing activities in a product line organization vary in scope from encompassing the entire product line to focusing on a specific product or even examining an individual component that is one part of a product. This wide range of testing activities addresses more complexity when compared with a typical single system development. Furthermore, the changes in core assets or extensions to the common platform of the product line, mostly occurring for unpredictable variability, might possibly affect the existing products. Therefore, regression test scope might dramatically increase as the product line scope evolves.

Software product lines make good use of modeling techniques like feature models or orthogonal variability model, for the representation of variability, but neither of these conventional models provide a complete solution for defining the regression test scope. For

overcoming the additional time and effort spent on regression testing, the relations between core assets and products based on variation points must explicitly be defined in order to correctly point out the impact of changes in core assets on the products. A new modeling technique, called Test Oriented Service and Object Model for Software Product Lines, is proposed in this thesis, which aims to reduce the regression test scope after developing new products in a software product line.

The decomposition of core assets into smaller indivisible unitary elements and variant based association of these elements with products constitute the underlying rationale of the proposed modeling approach. The model formally indicates the changes in the core assets with their relations to the products based on variation points when extensions to a product family require changes in the core assets of domain engineering. This new approach results in the opportunity of discarding the dispensable parts from the regression test scope.

The model is implemented on Loyalty and Campaign Management System (LCMS) which is being developed and maintained as a sample software product line. The product family of LCMS is composed of loyalty campaigns which are derived by using the core assets. The problem of determining the regression test scope after construction of a new campaign or modification of existing campaigns in LCMS is tried to be solved using Test Oriented Service and Object Model for Software Product Lines.

The organization of this thesis is as follows: After an introduction in Chapter 1, a general research on various aspects of software product lines is presented in Chapter 2. Variability in software product lines is described in Chapter 3 and testing process in a software product line is discussed in Chapter 4. Chapter 5 introduces the current system specification for LCMS. Chapter 6 proposes the Test Oriented Service and Object Model for Software Product Lines and presents the general principles of the model. In Chapter 7, as a case study, the implementation of the new modeling technique on LCMS including the adaptation of new business requirements is presented and reduced regression test scope for new features is determined. Finally, Chapter 8 concludes this thesis and presents the future work.

CHAPTER 2

SOFTWARE PRODUCT LINES IN GENERAL

Many approaches have been presented for software reuse over the years. However, none of them could provide a complete and effective solution for software projects being completed on time and within the budget. A new approach, called Software Product Line Engineering, supports large-grained intra-organization software reuse and has been highly challenging over the last few years. Although there might be some risks in changing the way doing the business, organizations can gain considerable business benefits by applying product line development approach.

2.1. What is a Software Product Line

In the early stages of software engineering, almost all software products were relatively small and simple. However, the situation has changed drastically in time and recently the size and complexity have increased for software products and other products having embedded software. Therefore, there is a strong need for product line engineering approach in software development. Product line engineering is not a new concept in manufacturing, or in other engineering areas. However, it is a relatively new paradigm in software engineering to develop software applications (software-intensive systems and software products) using platforms and mass customization [2]. In other words, a software product line can be defined as 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 [1].

Since the requirements frequently change even before the deployment of the product and many products are initiated from the existing ones instead of being initially developed, the classical development-and-then-maintenance model have been inappropriate for today's way

of software development [7]. Therefore, software product line engineering approach is likely to be the most effective way in terms of rapid development resulting in shortened time to market, reduced costs and increased product quality.

Reuse is the key concept for software product lines for decreasing costs and increasing quality. Almost all software development methodologies concentrate *opportunistic* reuse, but software product lines differ from the other methodologies because reuse is planned, enabled and forced, thus *predictive*. Software product lines reuse assets which are intentionally developed for reuse. Therefore, a software product line differs from a single-system development with reuse and also it is not a collection of releases and versions of single products. Besides, a software product line is not just a component-based development, just a reconfigurable architecture or just a set of technical standards. Overview of the basic structure involved in a software product line is shown in Figure 2.1.

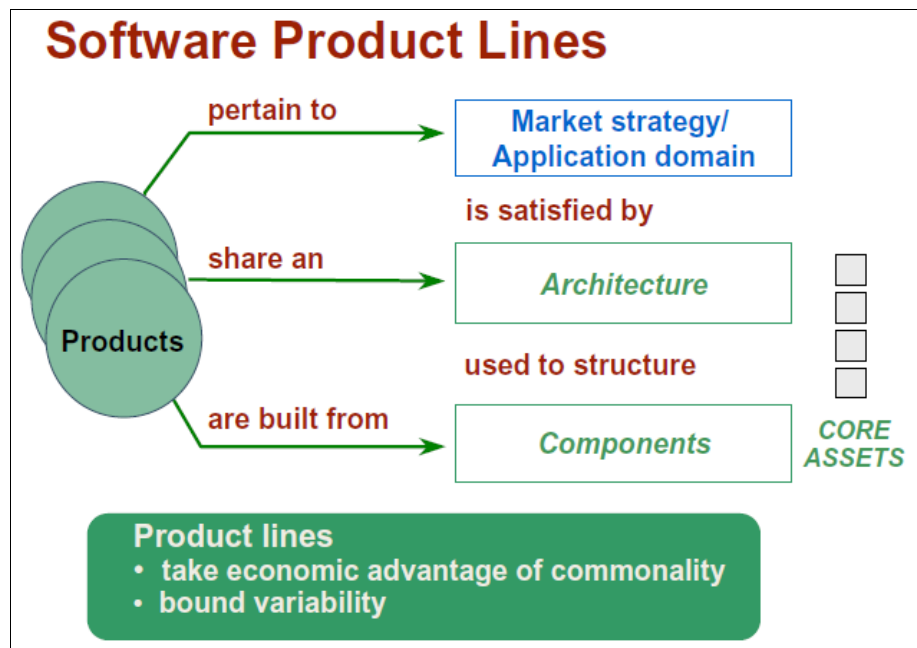


Figure 2.1: Overview of basics in a software product line [5]

2.2. Software Product Line Processes

The software product line engineering paradigm is based on the separation of the whole software development into two processes called *domain engineering* and *application engineering*. Figure 2.2 shows the framework for software product lines including domain and application engineering processes with their sub-processes and their interactions.

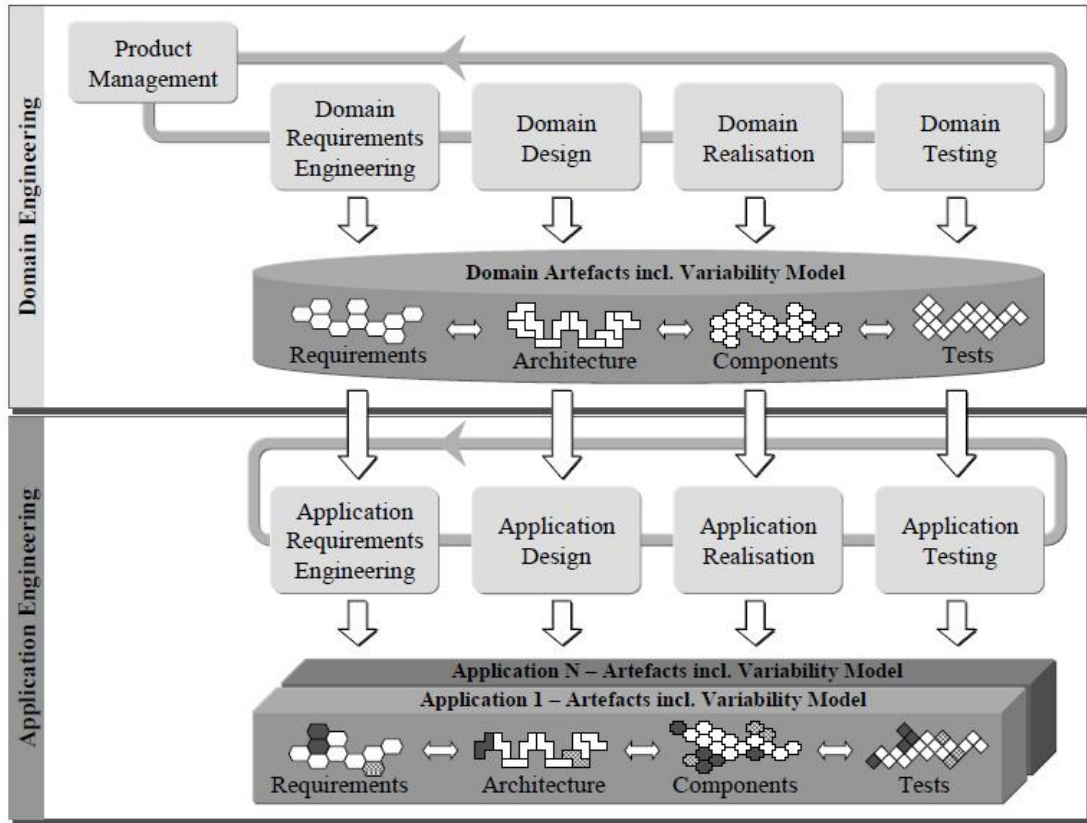


Figure 2.2: The software product line engineering framework [2]

2.2.1. Domain Engineering

Domain Engineering is the process in which the commonality and the variability of the product line are defined and realized [2]. This process deals with all types of core assets composing the reusable platform which is used to build up new products. The variability and commonality between products are mainly managed in domain engineering. The major aim of domain engineering is to define and construct reusable artifacts ensuring adequate variability and commonality for building different products. Also, domain engineering should manage the scope of the product line by defining the set of applications planned.

The sub-processes of domain engineering which are shown in Figure 2.2 are [2]:

- **Product Management:** The economic aspects and the market strategy of the product line are managed throughout product portfolio and scope of the product line.
- **Domain Requirements Engineering:** All common and variable requirements for the products are created and managed through elicitation, documentation, negotiation, verification/validation and management phases [3].

- **Domain Design:** The reference architecture is defined, providing a common, high-level structure for all applications.
- **Domain Realization:** The reusable software components are designed in detail and created after a make/buy/mine/commission decision [3].
- **Domain Testing:** The reusable software components are validated and verified.

The reusable development artifacts of domain engineering which are created in these sub-processes are [2]:

- **Product Roadmap** deals with the scope of the software product line platform and provides a plan for future development of the product portfolio. This artifact is the output of Product Management sub-process, but it is not shown in the framework picture in Figure 2.2 since it is not a software development artifact.
- **Domain Variability Model** defines the variability of the software product line in terms of variation points and variants as the output of all sub-processes of domain engineering.
- **Domain Requirements** are documented in natural languages or conceptual models and include all variable and common requirements for core assets of the product line as the output of the Domain Requirements Engineering sub-process.
- **Domain Architecture** determines the structure and the texture of the applications in the software product line as the output of the Domain Design sub-process.
- **Domain Realization Artifacts** are comprised of the detailed design and source code of the reusable software components and interfaces as the output of Domain Realization sub-process.
- **Domain Test Artifacts** include test plans, test cases and scenarios for domain realization artifacts as the output of Domain Testing sub-process.

2.2.2. Application Engineering

Application Engineering is the process in which the applications of the product line are built by reusing domain artifacts and exploiting the product line variability [2]. This process is responsible for combining the core assets, binding the variabilities and reusing the commonalities from domain engineering in order to build different products. The product-specific assets, which are not in scope of domain engineering, are also managed in application engineering.

The sub-processes of application engineering which are shown in Figure 2.2 are [2]:

- ***Application Requirements Engineering:*** All requirements are collected and analyzed to specify a certain product.
- ***Application Design:*** The product architecture is derived from the reference architecture.
- ***Application Realization:*** The considered application is created, i.e. the desired product is implemented.
- ***Application Testing:*** An application is proved to have sufficient quality and is satisfying the requirements by using verification and validation activities.

The reusable development artifacts of application engineering which are created in these sub-processes are [2]:

- ***Application Variability Model*** defines the bindings of variabilities and variability extensions for a particular application as the output of all sub-processes of application engineering.
- ***Application Requirements*** consists of all specifications for a particular application as the output of the Application Requirements Engineering sub-process.
- ***Application Architecture*** determines the overall structure for a particular application in the software product line as the output of the Application Design sub-process.
- ***Application Realization Artifacts*** are comprised of the detailed design of components and interfaces of a particular application and the executable product as the output of Application Realization sub-process.
- ***Application Test Artifacts*** include all test documents for a particular application as the output of Application Testing sub-process.

2.3. Software Product Line Management

In the previous sections, the two major processes of a software product line, domain engineering consisting of developing core assets and application engineering dealing with product development has been presented. Besides these two, there is one more essential activity for a product line, which is called management. The products are built from core assets, but also core assets may be built from existing products. The synchronization between

these activities is arranged by management. The three essential activities of a software product line can be shown in Figure 2.3.

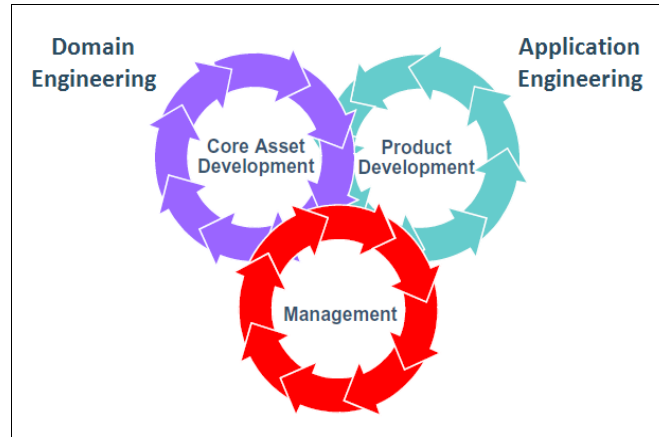


Figure 2.3: The three essential software product line activities [1]

Software product line management can be defined as the combination of technical management and organizational management. **Technical management** provides the cooperation and alignment of core asset and product development activities. This level of management ensures that the groups responsible for core asset and product development work together when necessary, they follow the defined processes of the product line and have a common knowledge base. **Organizational management** mainly concentrates on the organizational structure and resource allocation. This level of management also plays a critical role in the success of a product line by coordinating, supervising and training the resources, developing an acquisition strategy, managing external interfaces like customers and suppliers, and creating a product line adoption plan.

2.4. Software Product Line Practice Areas

In order to make a software product line functional, well defined and more detailed practices should be performed under the essential activities. These practices are categorized in different practice areas. A practice area is defined as a body of work or a collection of activities that an organization must master to successfully carry out the essential work of a product line [1]. In software product line engineering, the practice areas are organized in three categories:

- **Software engineering practice areas** apply the convenient technology to build and maintain both core assets and products.

- *Technical management practice areas* concentrate on the engineering aspects for building core assets and products.
- *Organizational management practice areas* deal with the orchestration of the whole software product line effort in terms of resources, business and funding.

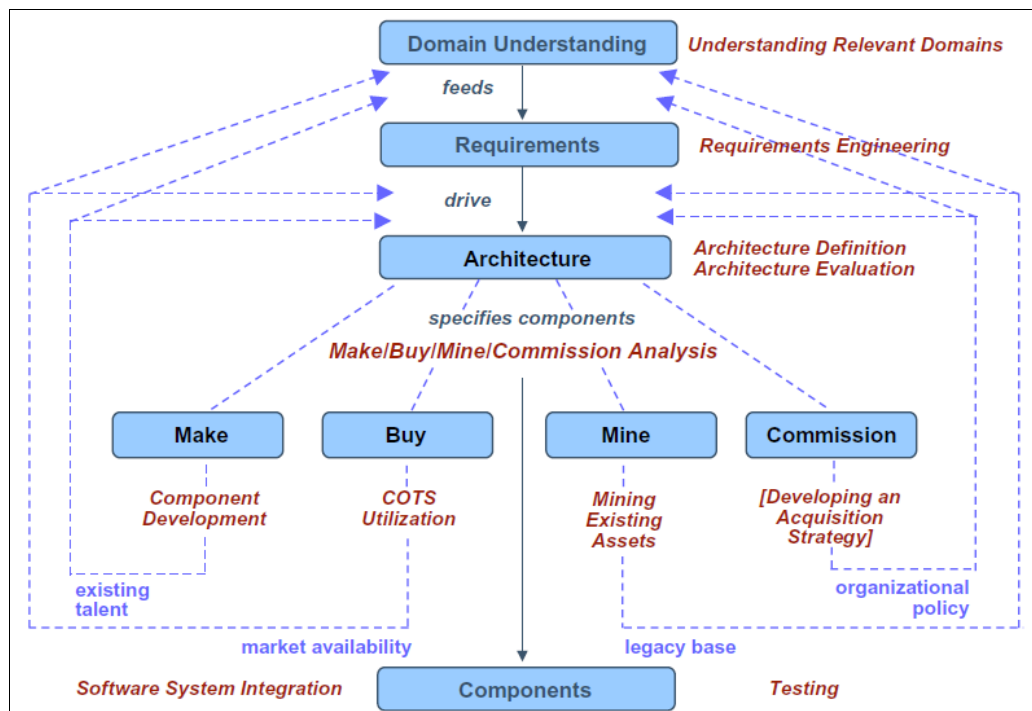
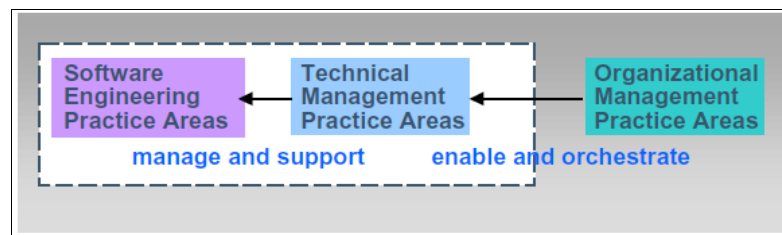
The practice areas under each of these categories [1] are listed in Table 2.1.

Table 2.1: Software product line practice areas

Software Engineering Practice Areas	Architecture Definition
	Architecture Evaluation
	Component Development
	Commercial off-the-shelf Utilization
	Mining Existing Assets
	Requirements Engineering
	Software System Integration
	Testing
	Understanding Relevant Domains
Technical Management Practice Areas	Configuration Management
	Data Collection, Metrics, and Tracking
	Make/Buy/Mine/Commission Analysis
	Process Definition
	Scoping
	Technical Planning
	Technical Risk Management
	Tool Support
Organizational Management Practice Areas	Building a Business Case
	Customer Interface Management
	Developing an Acquisition Strategy
	Funding
	Launching and Institutionalizing
	Market Analysis
	Operations
	Organizational Planning
	Organizational Risk Management
	Structuring the Organization
	Technology Forecasting
	Training

All these practice areas can be applied to both core asset development and product development activities. They all serve for achieving the efficient functionality of a software product line. It should be noted that, the practice areas mentioned above constitute a

complete set for a mature product line. An organization at the early stages of product line approach might not master all of these practice areas in the beginning, but it should have the intention to proceed in all in order to achieve the goal of a successful product line.



2.5. Transition to a Software Product Line

2.5.1. Transition Strategies and Approaches

Transition from traditional software development to software product line approach is not very easy for organizations in many aspects. External factors like the resistance of the development team, difficulty in understanding the product line paradigm, coarse and stationary structure of the organization, etc. may result in unpredictable time and effort for the transition. Some case studies have shown that some transitions facing those difficulties may require a huge effort of 2 to 5 years. However, there are also success stories which overcome the obstacles in a proper way and reduce the transition time to a minimal level of 2 months [6].

Some of the reasons for the large time and effort diversity in transition strategies may be stated as follows [6]:

- i. If software development artifacts are somehow reusable, this reduces the transition time and effort. The transition will be easier if this reusability comes from an existing library or even from re-engineering of an existing product, rather than building those artifacts from scratch.
- ii. If the initial state has similar products instead of developing a completely new one, the transition can be considered as an enhancement of a software product line. These existing products might have been developed by conventional techniques, but it is still better than creating a new product from the beginning. Artifacts from existing products can often be reused and re-engineered for enhancement of a software product line in order to save time and effort.

Although the time and effort were extremely high in the early times, there are recent enhancements in software product line transition strategies. So called *lightweight* approaches use the techniques below for lower costs of transition to software product lines [6]:

- i. In order to minimize the effect on the organization, processes, software and architecture, the differences between single-system and product line engineering is minimized.

- ii. An incremental adoption strategy starting from a small subset of assets, products or resources is implemented.
- iii. Existing software product line tools and techniques are used whenever possible.
- iv. Reactive approaches, in which transition starts with one or few products, are used in order to defer the effort required for all products.
- v. Development life-cycle is constructed so as to minimize the complex and costly merging and product-specific configuration management overhead.

The transition approaches, using different combinations of these techniques and having self advantages and disadvantages, can be presented as follows [2] [5]:

Reactive (Pilot-Project) Approach: Start with one or few products.

- From the initially developed few products, first core assets and then future products are started to be generated.
- Robust, extensible and appropriate architecture and other core assets should be created initially for future needs.
- The transition cost will be lower, but the scope of the product line will evolve dramatically.

Tactical Approach: Start with only some specific sub-processes and methods.

- Transition starts in problematic sub-processes partially and probably informally.
- After a short initial phase, the other complementary sub-processes and the plan for further progress should be developed in order to complete the transition.
- This approach concentrates on the urgent needs of the organization. But it has the risk to fail because of not forcing an overall transition plan.

Proactive (Big Bang) Approach: Build a software product line at once.

- First, domain engineering is performed completely.
- When the core assets are built, application engineering starts and products are developed using the core assets.
- The organization will not be productive until the transition is complete.
- This approach requires a predictive knowledge and well understanding of software product line paradigm, and upfront investment for future benefits.

Incremental Approach: Develop a software product line step by step.

- Initially, part of the core asset base including the architecture and components for early requirements are developed.
- Using the initial core asset base, develop one or more products.
- Develop part of the rest of the core asset base.
- Develop more products.
- Repeat developing the rest of the core asset base and developing new products.

Comparing the four approaches, the most effective way of adopting a software product line seems to be the incremental approach. It proposes a smooth transition which does not change everything at once. When incremental approach is applied, some part of the organization can start transition to a software product line, while the others still can continue developing software traditionally. This means, no drastic and sudden change in the whole structure of the organization.

Any of the key practice areas defined in Section 2.4 can be considered in an incremental transition. However, the most appropriate way is to start with the practice area in which inefficiencies or bottlenecks are likely to appear. After eliminating the highest inefficiency problem, then the next bottleneck in the sequence can be the subject for the next iteration in the incremental transition [8].

2.5.2. Organization Aspects

Transition to software product line methodology requires changes in the way of doing business, but only the changes in processes and technical management aspects will not be sufficient for success. The initiative for software reuse with product line approach should be supported by an appropriate organizational structure.

There are many factors affecting the decision for the right organizational structure for a company. Some of these factors can be stated as the market, history and the culture of the company, power distribution in the company, experience of the employees and practice of the organization. The organizational structures should be evaluated with regard to some essential properties [2]:

Decision making: Just a small - but sufficient - number of people from both domain and application engineering should be involved in decision making. This will help for efficiency and avoid from spending long time to make decisions.

Overhead time: The overhead time, which is spent for coordinating the work, should not exceed the time spent for effective work. The overlaps and dependencies between organizational units should be minimized.

Reflecting responsibilities: The explicitly assigned responsibilities for a position in the organization should reflect the implicit ones. For example, for a software product line, it is important that the organizational structure ensures the presence of domain engineering, application engineering and the coordination between them.

Motivation: The employees should be motivated and encouraged in an equitable way for their valuable contribution to overall success of the company. For example, there should be no difference in valuation of the staff working for domain and application engineering in a software product line.

Customer focus: Organizational units should never lose their customer focus whether they have direct contact with customers or not. In software product lines, precautions should be taken for especially domain engineering units which are usually not in direct contact with the end-users.

The structure of a software product line organization can be analyzed in two different ways. The first categorization of structures depends on the hierarchical construction of the organization. Another point of view is the orientation of the organizational units on either products or processes, or matrix organizations compromising orientation on both. The following two sections investigate the organizational structures for software product lines with respect to these two different aspects.

2.5.2.1. Hierarchical Organization Structures

The following hierarchical structures are suggested for handling the responsibility of the reusable assets within the organization, based on the size of the organization [9]:

Development Department: All software development is centered to and performed in a single department. Staff can be assigned to both core asset development and product development depending on the current needs of the organization as shown in Figure 2.6. No organizational specialization exists on either the domain engineering or application engineering. This model can be applicable in small organizations with up to 30 developers. If the number of staff members exceeds 30, some kind of organizational restructuring is typically required.

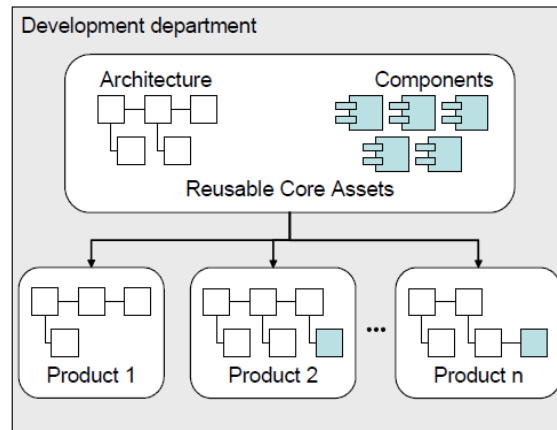


Figure 2.6: Development department model [9]

Business Units: The development team is divided into units that are centered and specialized on a specific product of the product line as shown in Figure 2.7. Each business unit is responsible for one or a subset of the products in the product line. Although this model is effective in its sharing and evolution of assets, the primary disadvantage is the absence of a unit within the organization which directly focuses on domain engineering. The business unit model is applicable to organizations with between 30 and 100 staff members.

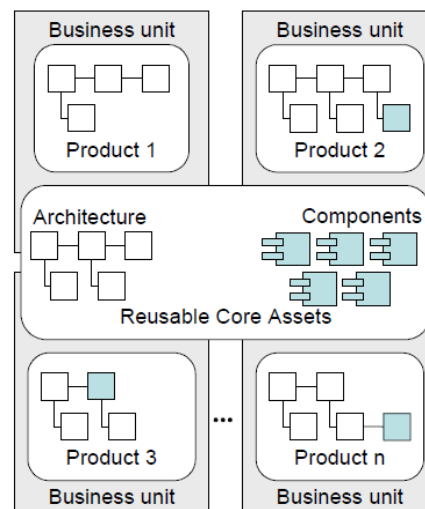


Figure 2.7: Business units model [9]

Domain Engineering Unit: This is the traditional and mostly suggested approach for software product line development. In this model, the domain engineering unit is responsible for the design, development and evolution of the reusable assets, i.e. the product line architecture and shared components that make up the reusable part of the product line as shown in Figure 2.8. In addition, business units, often referred to as product engineering units, are responsible for developing and evolving the products based on the core assets. The

general concern with this model is that the domain engineering group might lose focus of the customer requirements. However, organizations usually need a domain engineering group with core asset focus if the number of staff members exceeds 100.

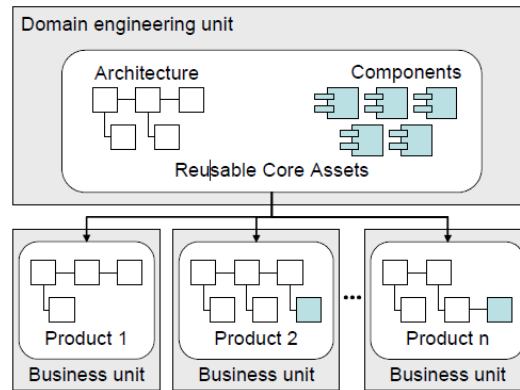


Figure 2.8: Domain engineering unit model [9]

Hierarchical Domain Engineering Units: This model includes several levels of product lines and domain engineering units as shown in Figure 2.9. If the variability and the number of products is very large or if the number of staff members exceeds several hundred it might be necessary to adopt this model. The more number of levels in this model increases the complexity for management and it can be assumed that an organization must be on a considerable process maturity level to be successful in this approach. It can also be added that if the scope of a product line can not be captured with this model, then the scope can be assumed to have been set too wide.

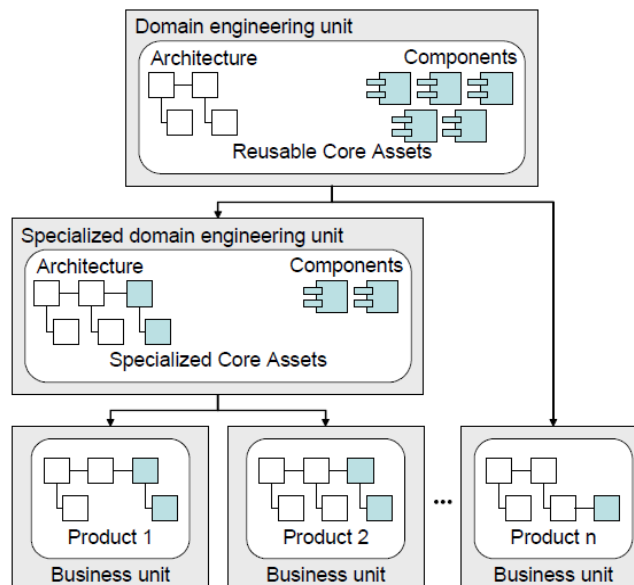


Figure 2.9: Hierarchical domain engineering units model [9]

2.5.2.2. Orientation-Based Organization Structures

The organizational model for software product lines can be classified into three basic structures according to their focus on products, processes, or a combination of both [3].

Product-Oriented Organization: The guiding principle for this model is the distinction of domain engineering and application engineering units. Usually, a separate unit for core assets and several units for product development is constructed and each unit have its own internal structure for development activities like requirements analysis, design and realization, as shown in Figure 2.10. This corresponds to the *domain engineering unit* model mentioned above as a hierarchical organization structure. If the scope of the product line grows and it becomes impossible to manage all core assets in a single unit, then domain engineering may be split into several units as in *hierarchical domain engineering units* model. The main advantage of the product-oriented organization is the ease of communication and interaction between closely related software engineering activities which are performed in the same unit.

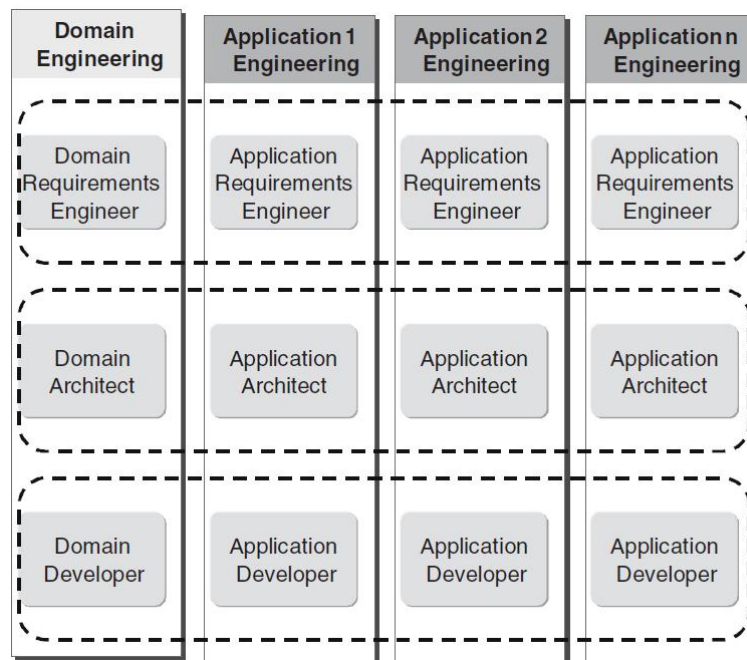


Figure 2.10: Product-oriented organization [3]

Process-Oriented Organization: The structure is set up on the software development activities rather than the products built through these activities. The organization units are constructed to perform specific phases of development as shown in Figure 2.11. The most important advantage of this model is the flexibility of assigning resources to both domain

engineering and application engineering. The developers themselves will be comfortable in building products during application engineering if they focus on reusability while developing core assets during domain engineering. This results in understanding usability as the primary notion of software product line engineering concept. On the other hand, the most common drawback in process-oriented structures is the governance of communication and collaboration among the organizational units participating in different development phases of the same application or product. So, this model best suits for relatively small organizations where communication is not likely a problem.

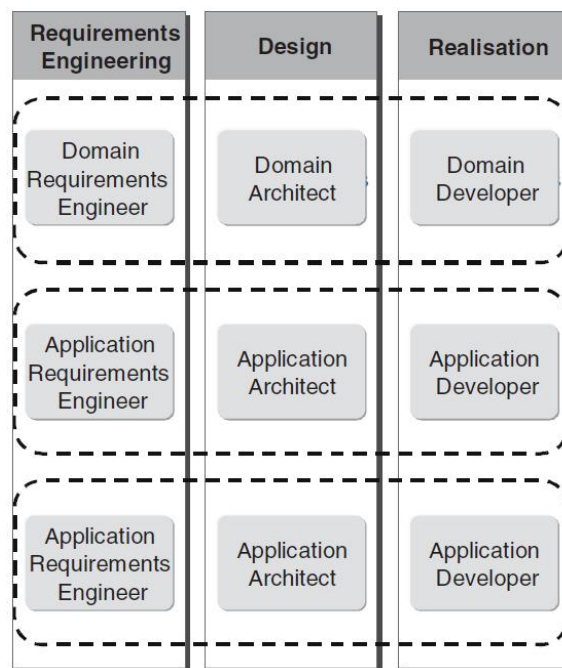


Figure 2.11: Process-oriented organization [3]

2.5.2.3. Matrix Organization Structures

For comprising both the needs for products and processes, which are two conflicting grouping criteria, matrix structures can be adopted. The matrix structure reflects the semantic perception of a software product line. Product orientation taking the customer's needs into consideration and process orientation with deep knowledge on how to do the work are combined in matrix structures. On the other hand, problems may arise in matrix structures because of the complexity of management and decision making difficulties in the crossing points. In the representation of a matrix organization, the application engineering products are aligned vertically as project units and the processes of product development are aligned horizontally as functional units. Domain engineering can be added to the matrix horizontally as a functional unit, vertically as a project unit, or separately outside the matrix [2]. Besides

application and domain engineering units, the other important activities like testing, asset management and product management should also be placed in the organization [3]. They are closely related to all other development activities and there are several ways of allocating these activities in the structure. An example matrix organization in which domain engineering is located as a project unit and testing is located separately as a functional unit is presented in Figure 2.12.

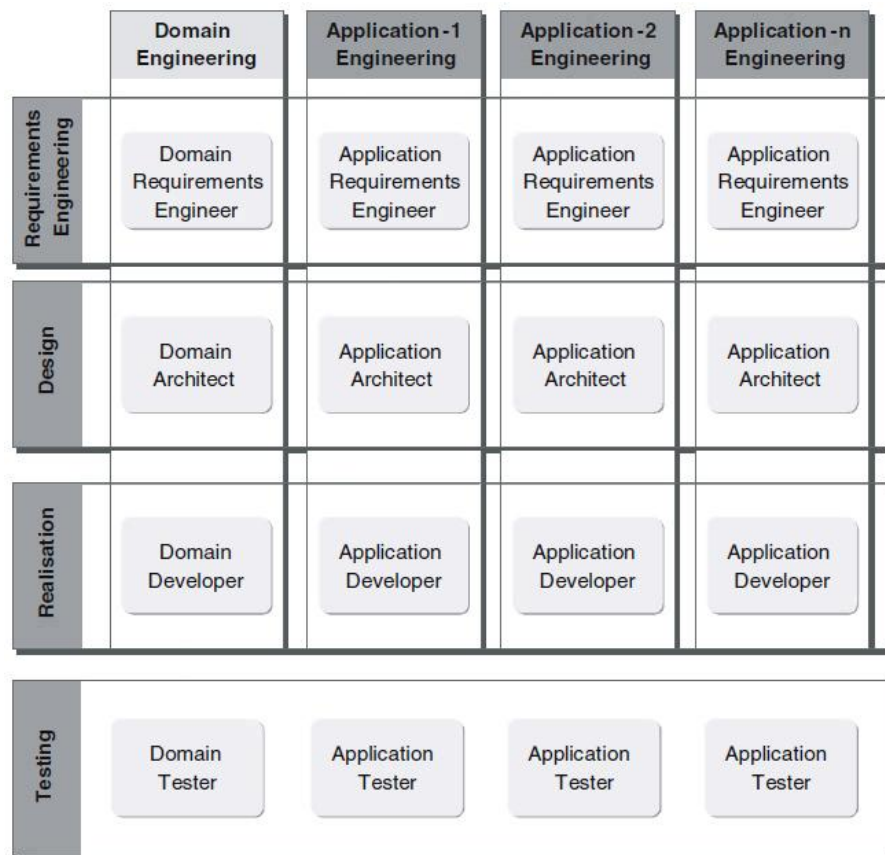


Figure 2.12: Sample matrix organization with testing as a separate functional unit

2.5.3. Software Product Line Maturity

Regardless of the transition approach or the structure of the organization, a software product line is supposed to evolve through a number of maturity levels which are shown in Figure 2.13. Brief descriptions of the maturity levels are as follows [10]:

Standardized Infrastructure: The infrastructure, on which the products are built, like operating system, database management system or user interfaces, is standardized.

Platform: A platform, consisting the standardized infrastructure and all common functionalities of the products in scope, is created.

Software product line: Functionality common to several but not all products becomes also part of the shared artifacts.

Configurable product base: The organization develops only one configurable product base, rather than developing a number of different products. The base is configured into a product at the organization or at the customer site.

Product Population: Shared product line artifacts are used to derive an extended set of products.

Program of product lines: Especially for very large systems, a software architecture is defined for the overall system whose components are configurable software product lines.

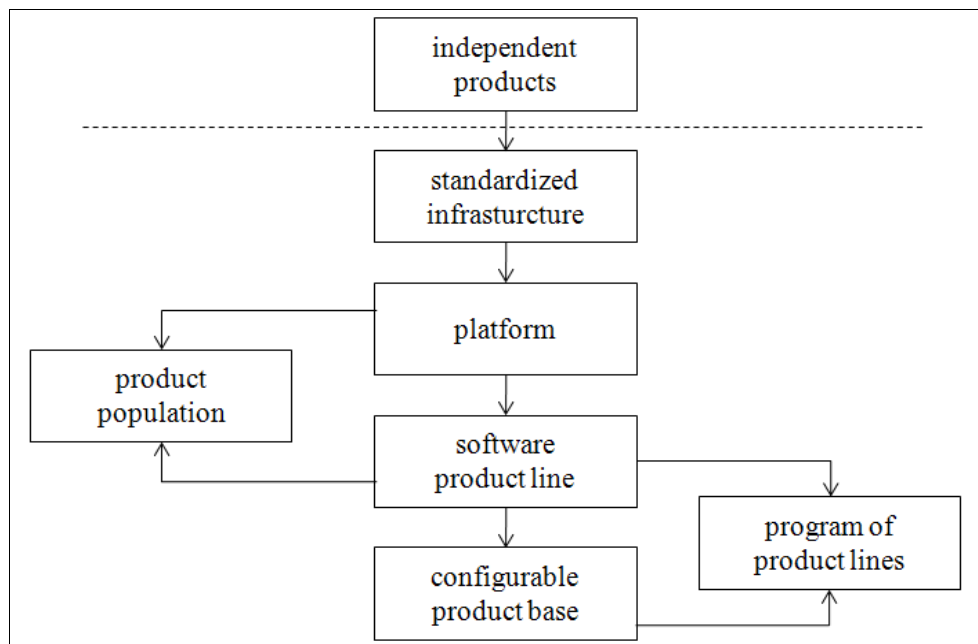


Figure 2.13: Maturity levels for software product lines [10]

In addition to the maturity levels applied to the overall software product line approach discussed above, different maturity levels can also be defined for the architecture, components and products which are the primary artifacts of the software product line [10].

Maturity levels for product line architecture:

- *Under-specified architecture:* The commonalities between the products are defined.
- *Specified architecture:* Both the commonalities and variabilities between the products are defined, but there might be product-specific changes in the common architecture.
- *Enforced architecture:* All commonalities and variabilities between the products are defined and all products share the common architecture.

Maturity levels for shared components:

- *Specified component*: The interfaces and specifications are defined for the product-specific components in the architecture.
- *Multiple component implementations*: There are multiple components which are shared by at least two products.
- *Configurable component implementation*: Only one highly configurable component is shared by all products.

Maturity levels for products:

- *Architecture conformance*: Products conform to the shared architecture of the product line.
- *Platform-based product*: Products share the components capturing commonalities only. Variabilities are handled individually by products.
- *Configurable product base*: All products are built from shared artifacts providing all functionality including both commonalities and variabilities.

Correspondence between the organizational structure and the maturity levels should be taken into consideration while a transition process is initiated. The organizational structure and the maturity levels for artifacts should be aligned with the applicable maturity level of software product line. Regarding the experience and best practices in the industry, Table 2.2 and Table 2.3 indicate the compatibility relation of software product line maturity levels to organizational structures and artifacts, respectively [10]. (Note that + stands for the full compatibility whereas + / - stands for partial convenience of intersections in the tables.)

Table 2.2: Compatibility relation of maturity levels to organizational structures

		SOFTWARE PRODUCT LINE MATURITY LEVELS					
		Standardized Infrastructure	Platform	Software Product Line	Configurable Product Base	Product Population	Program of Product Lines
ORGANIZATIONAL STRUCTURE	Development Department	+	+	+			
	Business Units	+	+	+			
	Domain Engineering Unit			+	+	+	
	Hierarchical Domain Engineering Units					+	+

Table 2.3: Compatibility relation of maturity levels to artifacts

			SOFTWARE PRODUCT LINE MATURITY LEVELS					
			Standardized Infrastructure	Platform	Software Product Line	Configurable Product Base	Product Population	Program of Product Lines
SOFTWARE PRODUCT LINE ARTIFACTS	ARCHITECTURE	Under-specified architecture	+	+			+	
		Specified architecture			+		+	+
		Enforced architecture				+		
	COMPONENTS	Specified component	+ / -	+			+ / -	
		Multiple component implementations		+ / -	+		+	+ / -
		Configurable component implementation			+ / -	+		+
	PRODUCTS	Architecture conformance	+					
		Platform-based product		+	+		+	
		Configurable product base				+		+

2.6. Benefits of Software Product Line Engineering

High quality products, quick time to market, effective use of limited resources, lower costs and mass customization result in improved efficiency and productivity, which are the key concepts underlying the universal business goals. Strategic software reuse through a properly managed product line plays a critical role in achieving these critical business goals. An initial investment is required for transition to a software product line, but the benefits in engineering, business and customer point of view will exceed the costs at the end.

Software product line approach improves efficiency and productivity of software development processes by:

- Achieving systematic reuse goals,
- Coping with complexity and evolution,
- Improving cost estimation,
- Reduction in the time and effort to develop and maintain a new product,
- Reduction in code size due to the removal of duplicated code,
- Increasing total number of products that can be effectively deployed and managed,
- And, enhancement of quality due to reduction in the number of defects per product.

The graph in Figure 2.14 illustrates the total engineering effort required for developing and maintaining a set of software products. The effective reuse of core assets and shared components in software product lines reduces the total effort compared to conventional software development. Moreover, if a lightweight transition strategy is applied with smaller up-front investment, the reduction in effort, i.e. cost, becomes more significant [6].

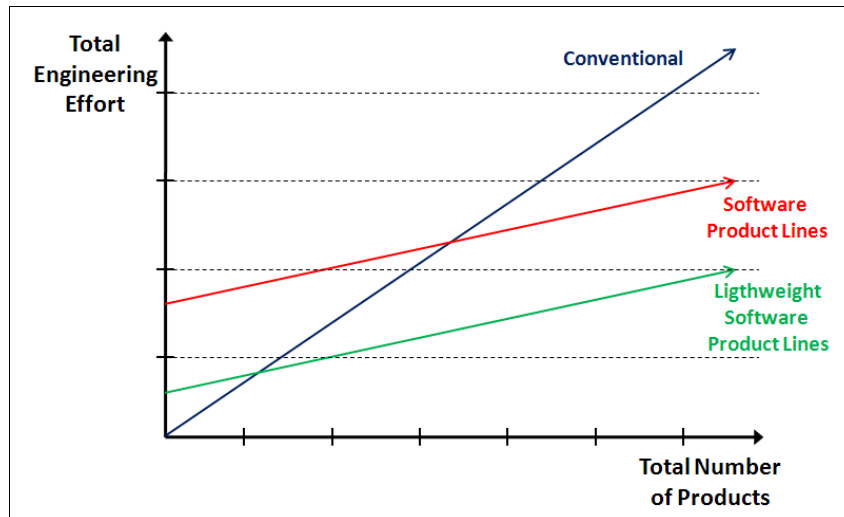


Figure 2.14: Reduction in development costs in a software product line [6]

In addition to effective reuse, the commonality and sharing in software product lines are important factors for quality benefits in terms of reducing the number of defects. Many products will take the advantage if a defect in a shared core asset is detected and fixed. Figure 2.15 illustrates the downward trend in the number of defects through consecutive releases of a particular product and a set of products [6].

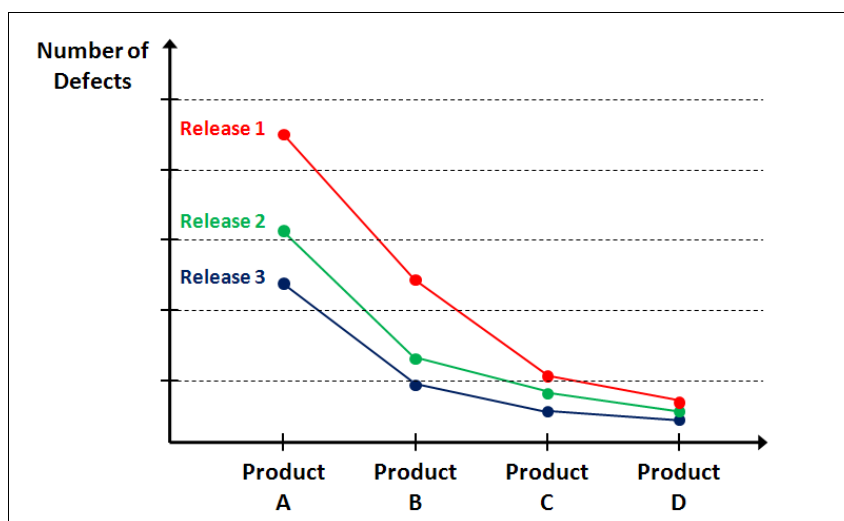


Figure 2.15: Enhancement of quality due to reduction in the number of defects [6]

The technical benefits of software product line approach discussed above bring out important impulses in business aspects and customer point of view. The major business gains of an organization coming along with a software product line approach can be listed as:

- Reduced time to market for new products,
- Better product quality and improved company reputation,
- Increased agility to expand into new markets,
- Maintaining market presence,
- Higher profit margins,
- And, improved competitive product value.

Besides, customer satisfaction is proved to increase due to:

- Common look and feel of products,
- And, higher quality with lower prices.

Among all the business benefits, the most critical success factor for a product is the time to market. In conventional single product development, the time to market is roughly constant for a particular product. In software product lines, although time to market seems higher initially because of building the common artifacts first, it is significantly reduced as many products are created in time, as shown in Figure 2.16 [2].

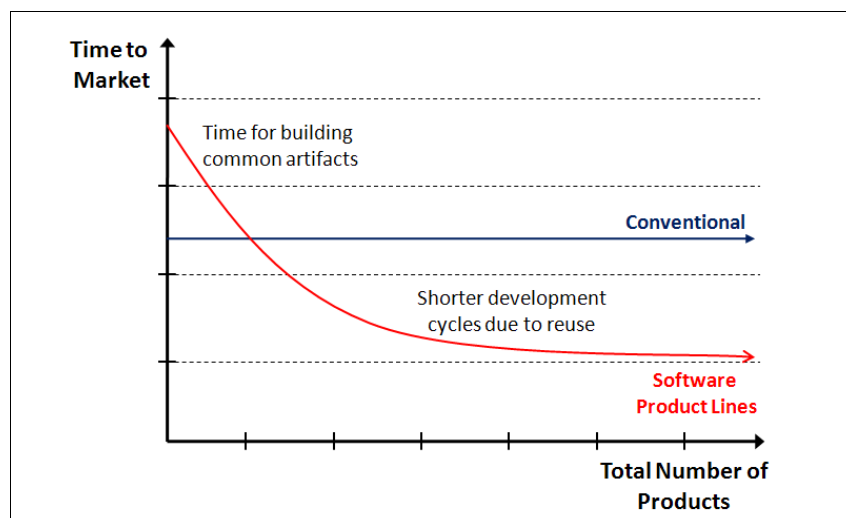


Figure 2.16: Reduction in time to market [2]

CHAPTER 3

VARIABILITY IN SOFTWARE PRODUCT LINES

A product line with no variability can be considered as a single system. The key concepts of a software product line are variability, the features that differ between some pair of products, and commonality, the features shared by a set of products. More formal and precise definitions for commonality and variability can be stated in terms of sets:

“A commonality is an assumption held uniformly across a given set of objects (S). Frequently, such assumptions are attributes with the same values for all elements of S . Conversely, a variability is an assumption true of only some elements of S , or an attribute with different values for at least two elements of S .” [11]

Besides commonality and variability among the products in a software product line, another important issue is handled in application engineering, which is called product-specific features. These are the characteristics which are only part of a single product. They need not be integrated into the product line framework, but the architecture should be able to support them [3].

3.1. Principles of Variability

Defining variability is the sum of all activities concerned with the identification and documentation of variability. Variability is defined during domain engineering and it is exploited during application engineering when appropriate variants are bound. In order to characterize variability in more detail, it is essential to define the terms variability subject and variability object. A variability subject is a variable item of the real world or a variable property of such an item and a variability object is a particular instance of a variability subject [2].

In software product line engineering, variability subjects and the corresponding variability objects are embedded into the context of a software product line and they represent a subset of all possible variations from the real world. These variability subjects and objects are necessary to realize a particular software product line. 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 [2]. Variations can be classified into three categories [12]:

Optional: A specific functionality of one product may not be contained in another.

Alternative: An instance from a set of alternatives can be selected for a specific property of a product.

A set of alternatives: Multiple instances of different alternatives can be selected for a specific product.

A variation point can be in three mutually exclusive states [13]:

Implicit: In the early phases of development there are many open design decisions which have not been deliberately left open so there is not a single point in the system that can be denoted as a variation point. These types of variation points are implicit.

Designed: The variation point is designed when the decision is left open intentionally.

Bound: When a decision is made for a designed variation point at a later stage, the variation point is bound to a variant.

The way of adding variants to the system can be predicted when a variation point is designed. Each variation point is associated with a set of variants that can be bound to it. In terms of the ability of adding new variants, a distinction is made between variation points as open and closed [13]:

Open variation points: New variants can be added.

Closed variation points: New variants can not be added.

Another important classification of variability is done according to the visibility of the variability to customers. Since customers want applications customized to their individual needs, they must be aware of at least a part of the variability of a software product line. On the other hand, variability is an integral part of domain artifacts and thus a major concern of the organization that develops the software product line. These two views are differentiated

by the terms external and internal variability. External variability is the variability of domain artifacts that is visible to customers, while internal variability is the variability of domain artifacts that is hidden from customers [2].

3.2. Binding Times

Each sub-process in application engineering binds the variability introduced by the corresponding sub-process in domain engineering. This has to be done in a consistent way to ensure that the required variant is built correctly. The moment of variability resolution in realization is often called the binding time of the variability. That is, the time at which the decisions for a variation point are bound is referred to as the binding time [6]. The design may intend moving the binding time to later phases in realization in order to increase flexibility. The trend to decide later on the binding time makes the binding time variable [2].

Examples of different binding times for software product lines include source reuse time, development time, static code instantiation time, build time, package time, customer customizations, install time, startup time and runtime [6].

A software product line can benefit from multiple binding times which allow some decisions to be bound earlier and others later in the lifecycle. With multiple binding times, the software product outputs from binding decisions at one production stage become partially instantiated software asset inputs for binding decisions at the next production stage as illustrated in Figure 3.1 [6].

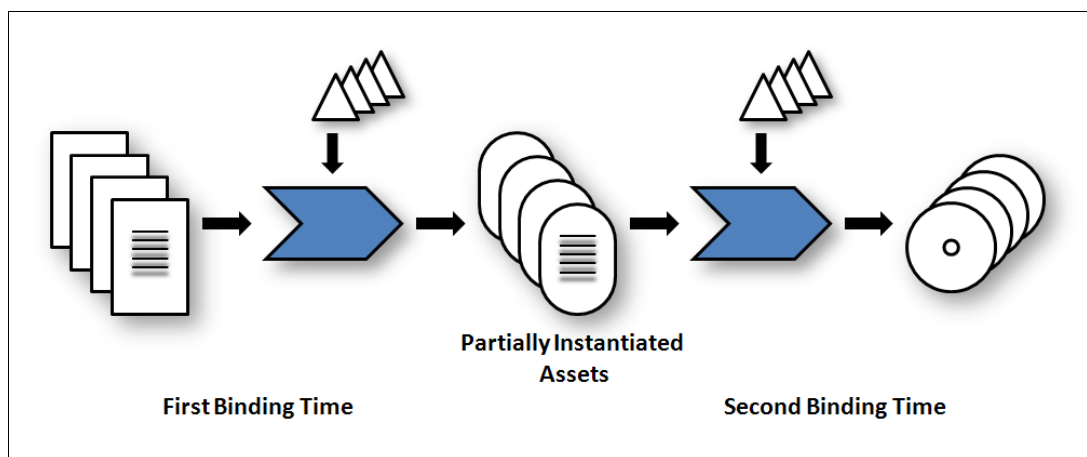


Figure 3.1: Partially instantiated binding times through development stages [6]

3.3. Variability in Time and Space

The fundamental distinction between variability in time and variability in space is essential for software product line engineering. Variability in time is the existence of different versions of an artifact that are valid at different times. Variability in space is the existence of an artifact in different shapes at the same time [2].

The evolution of development artifacts over time is an indispensable fact in software engineering since these artifacts have to be adapted to technological changes. This kind of change is denoted as variability in time which also applies to single system engineering. However, there is an important difference between single systems and software product lines in terms of variability in time. It is relatively easy to introduce changes in predefined locations identified by variation points in the domain artifacts of a software product line. Since the need for variation is recognized and introduced earlier, less effort is required for maintaining the requirements for changes in later phases of development [2]. Figure 3.2 illustrates the evolution of variability for the software artifacts in domain engineering and application engineering over time [15].

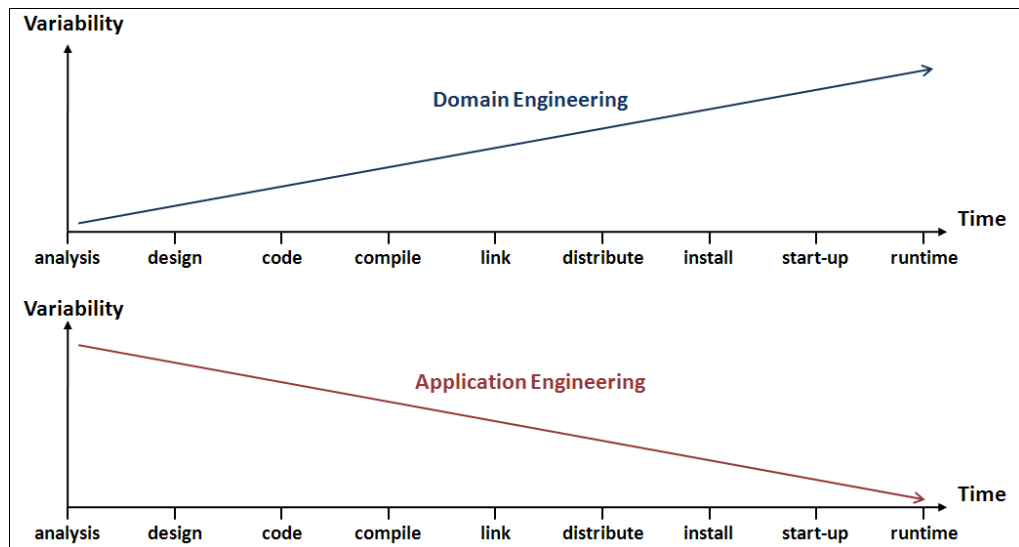


Figure 3.2: Variability in time for domain and application engineering [15]

Since single-system engineering does not focus on more than one product in a certain time, variability in time is relevant for only software product lines, in which the goal is building similar products that differ within a defined scope usually at the same time. Therefore – in contrast to single software system development – understanding and handling variability in space is an important issue of software product line engineering [2].

3.4. Levels of Variability

Variability occurs at different levels like product-line level, architecture level, component level, sub-component level, and the code level [16]. More precisely, variability points can be introduced at various levels of abstraction which are linked to different points in the lifecycle [17]:

- Architecture Description
- Design Documentation
- Source Code
- Compiled Code
- Linked Code
- Running Code

In Figure 3.3, the different transformations a system goes through during development are outlined. During each of these transformations, variability can be applied on the representation stating the level of abstraction subject to the transformation [17].

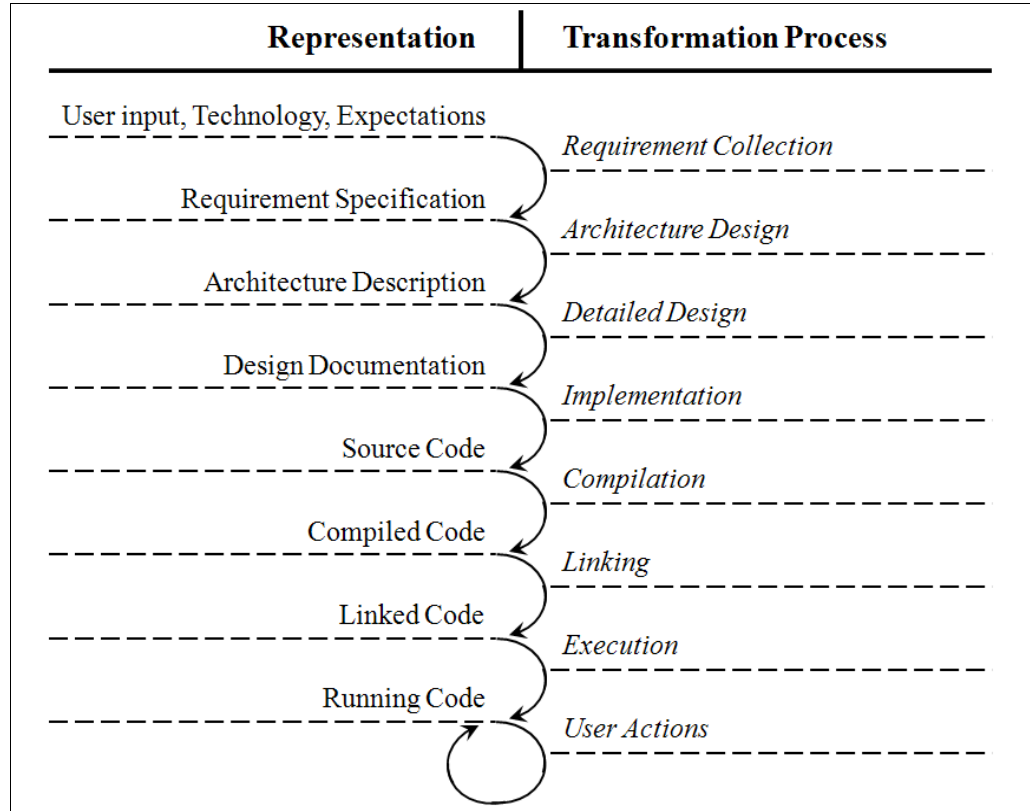


Figure 3.3: Levels of abstraction for variability [17]

3.5. Implementing Variability

A variability mechanism is a technique that enables automatic configuration of the variability in an application's requirements, models, implementation and test specifications. Variability in software product lines can be implemented in several ways during component development. Mechanisms supporting different types of variability are shown in Table 3.1 [4].

Table 3.1: Variability Mechanisms

Mechanism	Time of Specialization	Type of Variation Point	Type of Variant	Type of Variability
inheritance	at class definition time	virtual operation	subclass or subtype	Specialization is done by modifying or adding to existing definitions.
extension	at requirements time	extension point	extension	One use of a system can be defined by adding to the definition of another use.
uses	at requirements time	use point	use case	One use of a system can be defined by including the functionality of another use.
configuration	previous to runtime	configuration item slot	configuration item	A separate resource, such as file, is used for the specialization of the component.
parameters	at component implementation time	parameter	bound parameter	A functional definition is written in terms of unbound elements that are supplied when actual use is made of the definition.
template instantiation	at component implementation time	template parameter	template instance	A type specification is written in terms of unbound elements that are supplied when actual use is made of the specification.
generation	before or during runtime	parameter or language script	bound parameter or expression	A tool produces definitions from user input.

Variability mechanisms are used to automate the configuration of the applications of a software product line. Variability mechanisms in software product lines can further be classified in more detail at the code level as follows [14]:

- Aggregation/Delegation
- Inheritance
- Parameterization
- Overloading
- Properties
- Dynamic Class Loading
- Static Libraries
- Dynamic Link Libraries
- Conditional Compilation
- Frames

- Reflection
- Aspect-oriented programming
- Design Patterns

Several quality criteria have been described for evaluating variability mechanisms with respect to the construction of product line assets [14]. Some of these quality criteria can be listed as follows:

Binding time: The time at which the variability is bound to the asset, which can be at pre-compile time, at compile time, at initialization time, and at run-time.

Scope: The smallest entity of variability supported by the mechanism.

Flexibility: The binding times supported by the variability mechanism.

Efficiency: The overhead required to support the variability in the asset using the mechanism.

Separation of Concerns: The ease with which the variability and commonality in the assets can be decoupled using the variability mechanism.

Traceability: The ease with which the assets can be traced to the features and requirements of the software product line.

Modifiability or adaptability: The ease with which the assets can be modified during product line evolution using the variability mechanism.

Configurability: The ease with which the assets can be combined and configured for different application configurations of a product line using the variability mechanism.

3.6. Representation of Variability

A complete documentation of variability should at least include all the information needed to answer the following questions [2]:

What varies? The variable properties of the different development artifacts have to be explicitly defined and documented by variation points.

Why does it vary? The causes of external and internal variabilities should be defined. Stakeholder needs, laws, standards or product management decisions can result in external variabilities. Besides, the realization of an external variability or another internal variability can be possible causes of internal variabilities. The causes of all internal and external variabilities should be captured in textual annotations of variation points and variants.

How does it vary? The available variants should be explicitly documented and they should be linked to corresponding domain model elements by trace links which are called artifact dependencies.

For whom is it documented? The stakeholders may differ for variation points and/or its variants. For example, variability documentation for customers is different from variability documentation for software developers. This distinction is based on the different audiences for internal and external variabilities and should be explicitly distinguished in the documentation.

The three main advantages of explicit variability documentation can be listed as improvement of making decisions, communication and traceability. Decision making is improved by explicitly documented variability since engineers are forced to document the justifications for introducing a certain variation point or a variant. Providing a high-level abstraction of variable artifacts within explicit variability documentation improves communication about the variability of a software product line. Explicitly documented variability also allows for improved traceability of variability between its sources and the corresponding variable artifacts [2].

An important aspect of successful product line development is defining an architecture that enables systematic reuse and modeling the architectural details in order to explicitly represent the variability. A high level architectural representation of variability in a software product line can be introduced using the following basic notation [12]:

- Optional variant: There exists exactly one implementation that could be included in a product.
- Alternative variant: There exist multiple realizations of this variant and exactly one must be included in the product.
- Set of alternative variants: There exist multiple realizations of this variant and at least one must be included in the product.
- Optional alternative: There exist multiple realizations of this variant and one of it could be included in the product.
- Optional set of alternatives: There exist multiple realizations of this variant and a collection of it could be included in the product.

3.6.1. Feature Models

Many research contributions have suggested the integration of variability in traditional software development diagrams or models. Feature models are one of these approaches. Every variation in a product line context can be connected with a corresponding feature which supports variations for specific cases and conditions. Table 3.2 provides an overview of feature relation types and the inclusion criteria of a feature in a product line instance [14].

Table 3.2: Feature Relation Types

Relation Type	Meaning
Mandatory	The feature must be always included whenever its parent is included.
Optional	The feature is an independent complement that may be included or not.
Alternative	Only one of the alternative feature can be included in a product.
Or	A non empty subset of the features can be included in a product.
Requires	Whenever a feature, X, is included, another feature, Y, must also be included if X requires Y.
Excludes	If features X and Y exclude each other, no products can include both X and Y.

After the introduction of the feature models, a number of important extensions have been devised. For instance, in the original feature models, feature diagrams are allowed to be trees, while in some of the extensions they are allowed to be in the form of directed acyclic graphs. An important extension has been the introduction of the UML like cardinalities. Another critical extension has been the introduction of the attributes of features, which provide extra information about the features. These feature models are called extended feature models [26].

3.6.2. Orthogonal Variability Model

Modeling variability within the traditional software development models has some significant shortcomings. So, an orthogonal variability model that defines the variability of a software product line is proposed, which relates the variability defined to other software

development models such as feature models, use case models, design models, component models, and test models. The orthogonal variability model provides a cross-sectional view of the variability across all software development artifacts. The notational elements of orthogonal variability model which are shown in Figure 3.4 are as follows [3]:

Variation point: Description of differences that exist in the final systems.

Variant: The different possibilities that exist to satisfy a variation point.

Variability Dependencies: A basis to denote the different variants that are possible to fill a variation point. The notation includes a cardinality to determine the possible number of simultaneously selected variants.

Constraint dependencies: Description of dependencies among certain variant selections. There are two forms of constraint dependency:

Requires: The selection of a specific variant may require the selection of another variant (perhaps for a different variation point).

Excludes: The selection of a specific variant may prohibit the selection of another variant (perhaps for a different variation point).

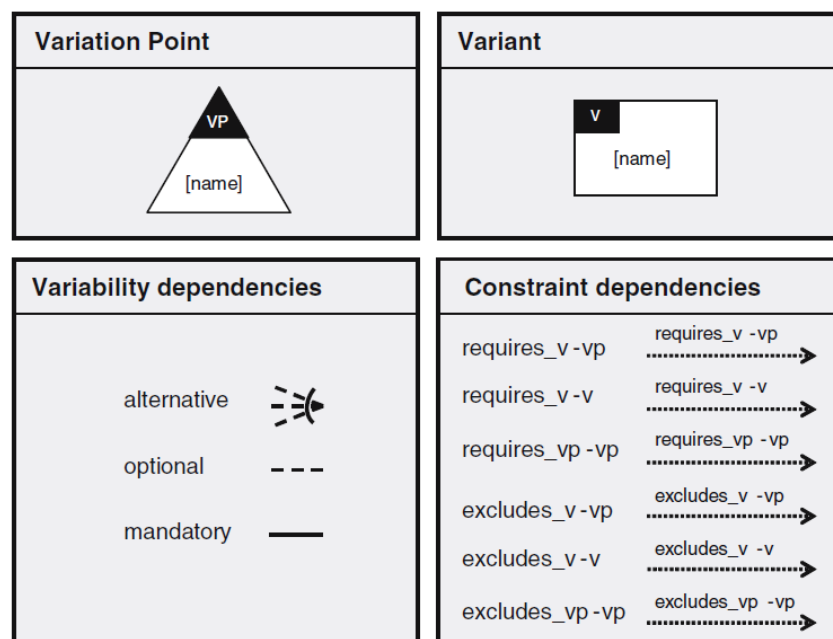


Figure 3.4: Graphical notation for orthogonal variability model [3]

CHAPTER 4

TESTING A SOFTWARE PRODUCT LINE

Software testing is a process which is used by a person or program to find out the quality, correctness and completeness of a software. An operation is done on the artifact under test and the result of the operation is compared with the expected result from the specification. Any irregularity of the expected result is called failure. Testing a program means to try to find out if there is any failure.

The success of a product line organization depends on a well organized and executed test process. Testing identifies the defects coming from the previous stages of the development lifecycle and ensures that the completed products fulfill the required qualifications and specifications. The test process should be designed to take the advantage of the methods used for scoping and scaling in a product line organization. Test-related activities should be sequenced and scheduled. Every construction activity should be followed by a testing activity which aims to verify and validate the output of the construction activity [25].

The degree to which a software owns a desired combination of attributes describes the quality of that software [23]. There are two major categories of quality attributes:

Observable by execution (operational): Performance, security, availability, usability, etc.

Not observable by execution (development): Modifiability, portability, reusability, integrability, testability, etc.

In a software product line, quality attribute requirements can be grouped as product line quality attributes which are related to application engineering and domain relevant quality attributes which are related to domain engineering. Product line quality attributes are considered development attributes or non observable by execution, whereas domain relevant quality attributes are usually operational or observable by execution [23].

Product line quality attributes are usually specific to application engineering where a set of related existing and future products are considered. They are related to variability or flexibility. Assessing the variability of a product line ensures that it is possible to get all the functionality of the products in the scope. Variability also ensures modifiability that allows evolution over time and configurability in the scope to get a set of related products.

Domain relevant quality attributes are usually addressed in domain engineering in a software product line. Due to the different quality requirements of products like different levels of security or performance expectations, the related assets in the domain engineering should ensure the assessment of those quality attributes for all products in the product line.

Inadequate testing will result in low software quality. The major risks that may rise by not doing enough or efficient testing include the following [1]:

- Insufficient unit testing will result in low quality of components.
- Lack of adequate testing tools will increase the effort for reaching an acceptable level of coverage.
- Inadequate specifications make it difficult to design testing activities.
- Insufficient integration testing will cause longer construction times for products.
- The expected level of reuse of test assets will not be realized if sufficient resources are not dedicated for testing.

4.1. The Testing Context

Testing in the context of a product line includes testing the core assets in domain engineering, the product-specific assets in application engineering, and their interactions. Testing is managed within the context of the other phases in the lifecycle and the testing activities are related to the construction activities in the development process. Appropriate test techniques are selected for each specific development process. Figure 4.1 shows that each testing activity immediately follows the related construction activity. Each testing activity should be designed not only to validate the output of the previous construction activity, but to identify the defects that escaped from the earlier test activities also. All these testing activities define a test process [25].

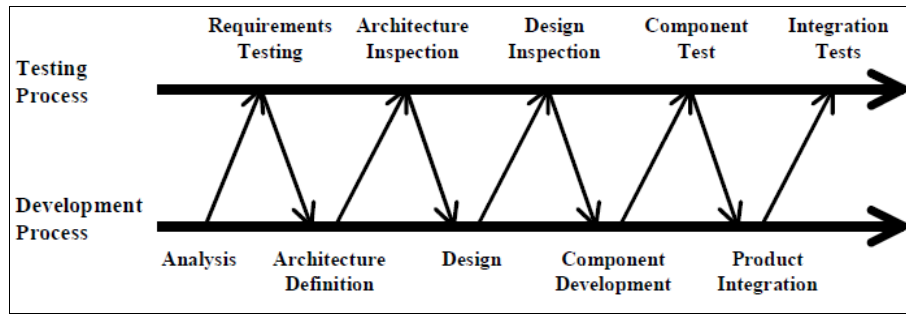


Figure 4.1: Test and development processes [25]

Product line principles should be applied to testing practices in the same manner as they are applied to development practices. This approach in test implementation, which is presented in Figure 4.2, lets testing benefit from the characteristics of product line engineering. During domain engineering, tests for core assets should be prepared simultaneously to the assets themselves. These test assets feed a test infrastructure. During application engineering, assets from this test infrastructure should be reused to lower the time required for testing and consequently for overall product development [22].

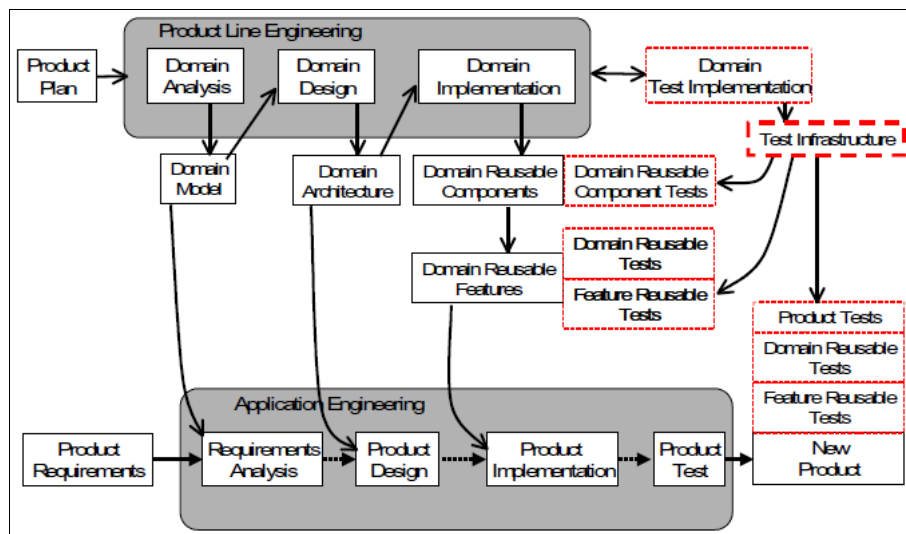


Figure 4.2: Test implementation in product line engineering [22]

Product line testing then can be done at three levels [22]:

Component Level: Traditional test techniques can be applied to execute component tests.

Feature Level: After the components have passed their component level tests, integration of these components should start with feature level validation.

Product Level: After the components and their integration are successful, the whole product should be validated against requirements.

Software product line testing should be designed for sharing testing core assets such as test cases, test plans, test tools, testing data, test scripts and testing reports, etc. All these assets should be reusable throughout the entire product line for all products instead of testing each similar application as an independent product. In another words, software product line testing is a reuse-based test derivation for testing specific products within a family of related similar products [19].

4.2. Software Product Line Testing vs. Single System Testing

In a single system development, the system is validated if the specific product operates correctly. However, the validation of a software product line is completed when every instance of the product line is assured to work correctly. The testing process for a single system should be expanded to address the domain and application engineering processes of a software product line. The test item, test design and test procedure documents can be extended during domain engineering, and then customized for a target application during application engineering [21].

Testing in a product line organization must examine the core assets software, the product-specific software and the interactions between them. Unlike single system development projects, responsibilities for testing may be distributed across different parts of the organization and testing represents an activity whose efforts are reused across a set of products. In order to take the full advantage of the benefits of reuse, planning is necessary. Critical success factors for software product line testing include structuring the testing software for reuse, including architectural support for testing, reusing assets for system integration testing, performing regression tests, and keeping track of acceptance tests [1].

There are two test processes as domain testing and application testing in software product lines and test activities are distributed between these two processes [2]. The main difficulty of domain testing is that there is no single, executable configuration of components that can be tested. Appropriate strategies are necessary for both ensuring early validation of the product line in domain engineering and achieving planned reuse of test artifacts by application engineering. Variable test artifacts provided by domain testing help in saving considerable effort since they are not created from scratch for each application. Variability in test artifacts originates from the variability introduced in requirements, design, and realization. But additional variability which is specific for testing artifacts like the test execution environment or test documentation should also be taken into account.

Similar to single system testing, ensuring an acceptable quality of a product is the goal of application testing in software product lines. This goal requires executing a set of tests that satisfies a certain coverage. Difference from single system testing arises at the point that the product to be tested is created partly during domain engineering and partly during application engineering. This brings out some repetition for testing the requirements and components which are identical in domain engineering. However, newly created test cases should be developed for application-specific artifacts. Besides, application testing should validate the variability binding for the new application complies with both the requirements specification and domain restrictions which are not an issue in single system engineering. Although some application-specific features require creating new test artifacts from scratch, most of the test artifacts come from domain testing or previous application testing activities, which can be reused after binding appropriate variants.

The variability of the product lines also requires variable test artifacts to ensure reusability of the tests. In a single system product the tests are static assigned to the existing components. A software product line member must be transformed from a chosen set of features and their related components before starting the test process. If a variant of a component is created, the responsible development team has to change the assigned test set to get it to work with the new variant.

In contrast to single-system engineering, testing activities in product line engineering have to consider product line variability as well as the differentiation between the two development processes. In order to achieve the required quality in a software product line, several test strategies like Brute Force Strategy, Pure Application Strategy, Sample Application Strategy or Commonality and Reuse Strategy can be applied [2].

4.3. Creating Reusable Test Artifacts

In software product lines, new products can be developed by reconfiguring the existing components on a predefined platform. But a component working correctly for one product is not guaranteed to work correctly for all others after being reconfigured. Even worse, the reconfiguration of components or changes in environment for new products may have negative effects on previously working ones. It is obvious that the components and related variants need to be tested individually when they are first created, but this is not enough. Since they may fail to provide their services due to residual defects, wrong usage or environment, they have to be tested again each time they are deployed to a new product [24].

A solution for this problem can be the strategy of reuse of core assets in software product line testing which can reduce testing effort during development, improve software quality, and potentially decrease the time-to-market of products and services [19].

Reuse in test artifacts depends on the key concept of variability in software product lines. Using the same variability mechanism for both development and testing is an effective and efficient approach that provides a correspondence between the product and its tests. This correspondence enhances the maintenance of the test assets. Using the same variability mechanism for implementing the variation in the product component and the test component makes both the components equally modifiable [20].

The test artifacts are produced by various testing activities, and some of them can be considered as core assets of the product line. These artifacts include test plans, test cases, test reports, test data, test software and test scripts. Test artifacts should be managed and their versions should be controlled under a certain configuration management system, in much the same way with the other development artifacts [25].

Test artifacts should be defined and structured in a way that they are reusable and modifiable as to be core assets. Proper definition of standardized and customizable non-executable artifacts containing test plans, designs and reports improve efficiency in the testing process. Executable test artifacts such as test cases and test scripts become also core assets if they are designed to permit the principles of variability. They have to be closely related to the code they intend to test, i.e. they have to be associated with the corresponding product development assets [25].

4.4. Domain Testing (Testing Core Assets)

Testing concepts are applied to domain engineering in two ways. First, testing itself produces reusable core assets like documents, test data sets and test software. Next, software core assets like components and non-software core assets like requirements model, analysis model, architecture, and detailed design should be considered. [1]

For testing software core assets in domain engineering, the test cases for domain artifacts are available and executed. The rest of the tests which are not covered by domain testing are executed in application testing. Although there is no single and complete executable

application to test in domain engineering, the typical five activities of the test process still can be performed in domain testing:

Test Planning: Test plans should be prepared based on domain artifacts like domain requirements, architecture, design and variability model of the product line.

Test Specification: Creation of reusable test cases is aimed.

Test Execution: Test cases are applied to the related domain assets and defects are corrected.

Test Recording: Documenting the test execution makes the tests repeatable and the test results verifiable.

Test Completion: The test record is analyzed and the error classes and the origins of errors are determined.

For testing non-software core assets in domain engineering, static test methods like inspection and evaluation can be applied. These inspection and evaluation activities result in several challenges for existing testing techniques [18]:

- Unit testing needs to distinguish among standard, optional and variant components.
- Integration testing needs to consider two different levels of integration on the overall architecture configuration and the individual products.
- Conformance testing of a product and its code can reuse information produced at the architecture level.
- Regression testing two different implementations of the same product architecture can be realized by applying techniques already proposed for product based regression testing.
- The information produced by testing a product in the architecture can be reused in order to test other products, using a development-level regression testing technique.
- Information used to test the implementation of a certain product in the architecture can be reused in order to test the conformance of another implementation with respect to its product.

Table 4.1 briefly describes the static testing techniques and the levels of coverage for non-software core assets in domain engineering. Increased coverage results in increased detection of defects [25].

Table 4.1: Static testing techniques for non-software core assets

Asset	Test Technique	Coverage Measure
Requirements Model	<p>Inspection by a team of domain experts who have not participated in developing the requirements.</p> <p>The team develops a set of scenarios that define its visions for the system.</p>	<p>1. Every use case should be touched by at least one of the expert's scenarios.</p> <p>2. Each variation point is sampled with multiple scenarios.</p>
Analysis Model	<p>Inspection by a team of domain experts who created the requirements and designers who will use the analysis model as input to architectural design.</p>	<p>1. One test scenario for each use case's default "usual course".</p> <p>2. One test scenario for several highly probable variants of the use case's "usual course".</p> <p>3. Test set expanded to include test scenarios for the use case's alternative and exceptional course.</p>
Architecture	<p>Inspection by a team of analysts who created the analysis model and designers who will use the architecture model as input to detailed design.</p> <p>An executable model may be used instead of a manual inspection if it is available.</p>	<p>1. One test scenario for each use case's default "usual course".</p> <p>2. One test scenario for several highly probable variants of the use case's "usual course".</p> <p>3. Test set expanded to include test scenarios for the use case's alternative and exceptional course.</p> <p>4. One test scenario for each identified architectural quality.</p>
Detailed Design	<p>Inspection by a team of architects who created the architecture model and developers who will code the interface implementations.</p> <p>The quality scenarios are used to guide a more in-depth analysis of the design.</p> <p>A syntax checker can be used if the Object Constraint Language or other parsable specification language is used.</p>	<p>1. One test scenario for each use case's default "usual course".</p> <p>2. One test scenario for several highly probable variants of the use case's "usual course".</p> <p>3. Test scenarios for the use case's alternative and exceptional course.</p> <p>4. Test scenarios for architectural qualities are re-analyzed.</p>

4.5. Application Testing (Testing Products)

Application testing aims to ensure the products properly meet the requirements. The quality and the functionality of the product is validated and verified by complementing the test activities performed in domain testing and reusing domain test artifacts. Unit test, integration test and system test are the levels of application testing and they are associated with application realization, design and requirements analysis phases in the development lifecycle, respectively. The results of each test level provide feedback for the related development activity. Figure 4.3 shows the information flow between application test levels and the related development sub-processes [2].

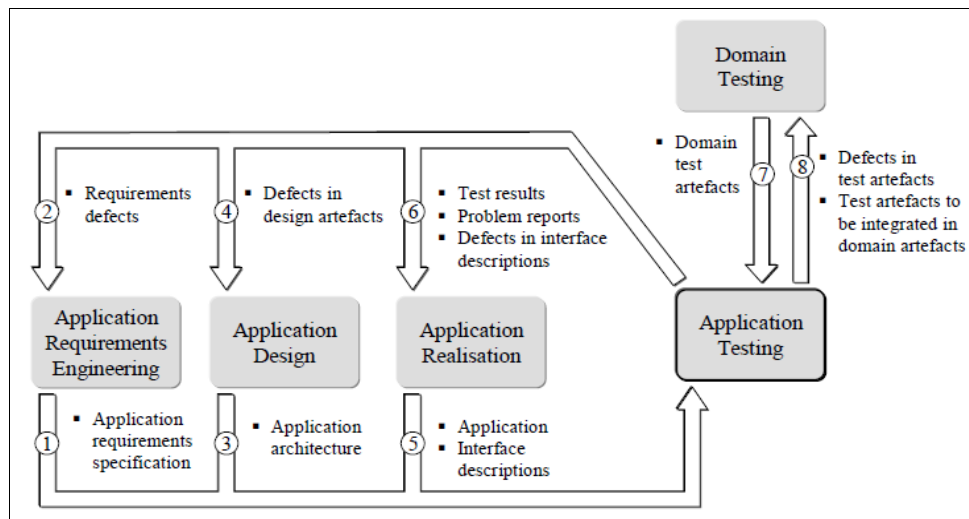


Figure 4.3: Information flow on application testing [2]

Application test artifacts are usually derived from domain test artifacts according to the guidance of the application test process description. The basic five activities of the test process can be performed in application testing as follows:

Test Planning: A product-specific application test plan is prepared.

Test Specification: Logical test cases, detailed test cases and the respective test case scenarios are created for the application.

Test Execution: The test cases are performed on the application and defects are.

Test Recording: The results of the execution are recorded.

Test Completion: The test record is analyzed for determining the error classes and the origins of errors. Besides, the detected errors are reported to the associated development sub-processes.

Application test process should be supported by tests related to variability and commonality which are the key concepts of software product lines. The bindings for variants and component configurations should be checked for correctness in terms of variability by performing the following two types of tests [2]:

Variant Absence Test: An application should not include any variants which were not defined to be included in that application.

Application Dependency Test: The application should be in conformance with the constraint and variability dependencies specified in the domain and application variability models.

Variability is not also an issue to be tested itself, but also has influence on different levels of application testing [2]:

Application Unit Test: All single components should be validated that they work properly for all possible combination of variants.

Application Integration Test: The interactions between common components, bound variants of variable components and application-specific components should be validated.

Application System Test: Although predefined domain system test artifacts can be used for commonalities, system test cases for application-specific variants should still be executed. The system test coverage can be enhanced by applying different types of requirement based tests like *Application Commonality Tests* for reused common artifacts, *Application Variant Tests* for reused variable or adapted requirements and *Application-Specific Tests* for new requirements.

CHAPTER 5

CURRENT SYSTEM SPECIFICATION

Today's competitive market in banking, especially in issuing credit cards, requires introducing new card brands and loyalty programs for both acquiring new customers and keeping the existing ones. Credit cards are also very effective instruments for banking in terms of cross-sell opportunities. Banks, especially the leader ones in the finance sector, persist on assuring the loyalty of individual customers and make important investment on loyalty and customer relationship management programs.

Loyalty and Campaign Management System is an important project for achieving the loyalty goals of banks. The basics and general structure of LCMS as a software product line will be presented in this chapter.

5.1. Loyalty and Campaign Management System

Loyalty and Campaign Management System is a multi-organizational, real-time & online core system for loyalty and campaign management in banking domain which can be used by various transactions coming from different distribution channels. It is completely designed, developed and still being maintained by software product line engineering methodology. Although it can be highly customized for the requirements of a specific bank, it can still be used in a multi-organizational way, i.e. use the products of LCMS can be shared and used by other banks which are business partners of that specific bank.

The scope of LCMS comprises all loyalty campaigns especially for credit cards. The reward pool management and complementary services supporting campaigns are also in scope of LCMS. Besides credit card campaigns, LCMS provides a general campaign management system for several banking operations and reward mechanisms for debit cards as well.

Regarding the technical specifications of the application, the architecture is distributed over different platforms. The core campaign modules serving different legacy transactions are COBOL programs running on IBM mainframe Z/OS, taking the advantage of power, robustness and high performance of mainframes. Campaign definition and management interfaces are Java and JSP applications running on UNIX/WAS servers and operating on the database through JDBC. Some application services run on different platforms specific for distribution channels like internet banking, call center or branches. The database is also on mainframe IBM DB2 for Z/OS and the data model consists of 183 relational tables. The overall application size including only the core components and applications specific to LCMS and excluding the distribution channel integrations are shown in Table 5.1.

Table 5.1: Application size of LCMS

Type	# of Files	Lines Of Code
Cobol Programs	147	118.968
Cobol Copy Books	71	4.848
Cobol Declaration Files	183	8.767
Job Control Language (JCL) Files	130	18.535
Java & JSP Programs	695	138.406

The products provided by LCMS are integrated to many distribution channels. All channels have their interfaces on different platforms, but they all use the core services of LCMS. Figure 5.1 shows all the distribution channels which are integrated to LCMS and Point of Sale (POS) integration is shown in Figure 5.2 as a sample channel integration.

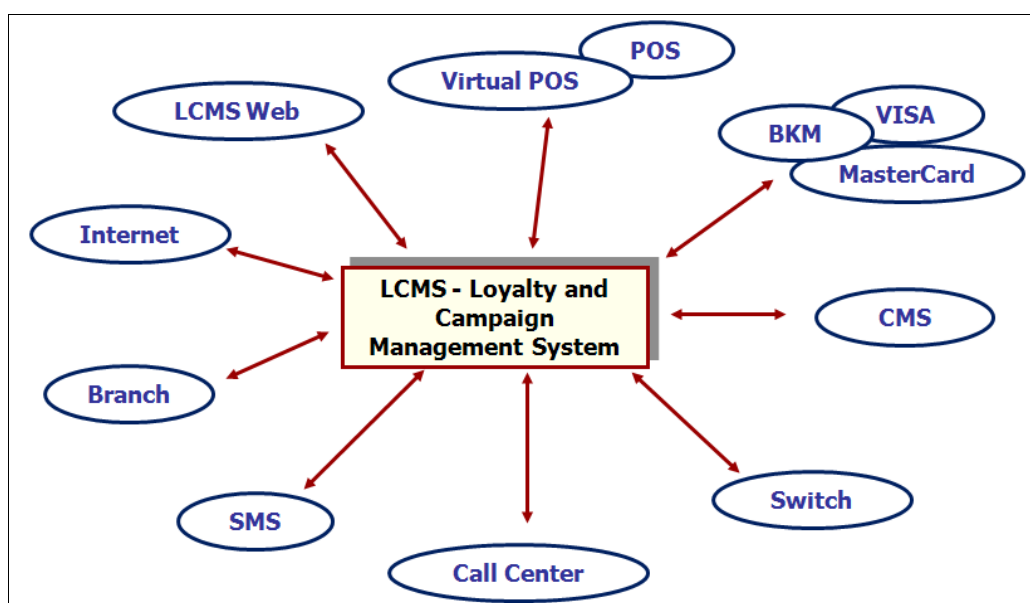


Figure 5.1: Distribution channels integrated to LCMS

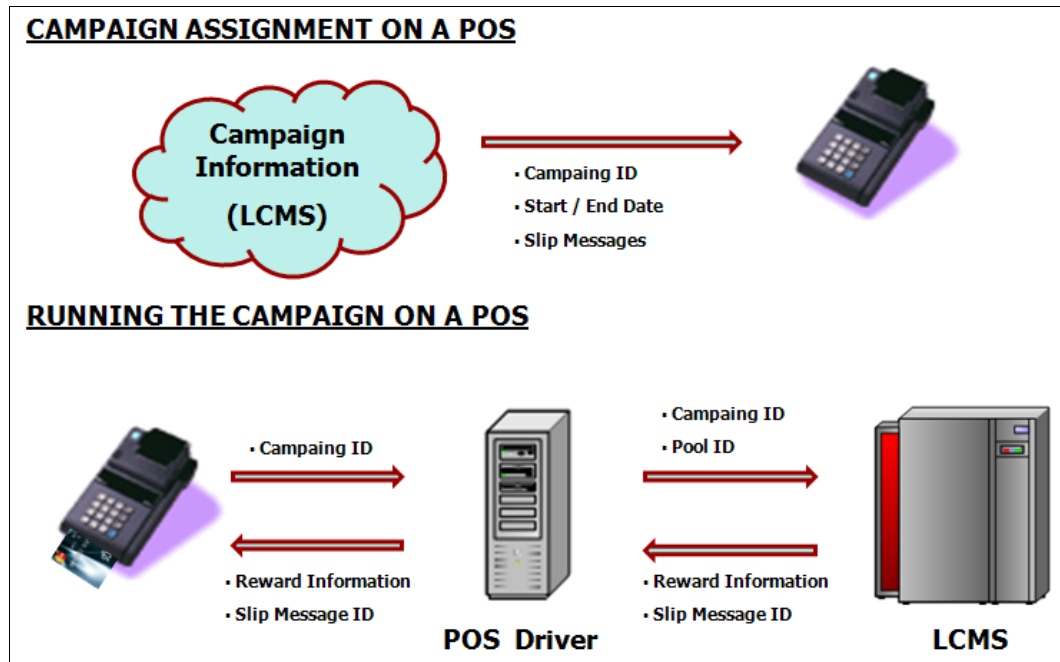


Figure 5.2: Sample channel integration of LCMS

5.2. Evolution of LCMS as a Software Product Line

The products of LCMS were developed iteratively in an incremental manner. Starting with the basic core assets in domain engineering, most critical campaigns and services were developed first and the overall system was constructed incrementally in time. As the system was growing, special care has been taken for the sake of product line engineering principles.

5.2.1. Organizational Structure and Processes

During the initial phases of LCMS project, the organization was process oriented. Responsibilities for different processes in the lifecycle were distinguished between process oriented units. The requirements analysis was performed by the System Development Team, whereas the Software Development Team was responsible for design and realization. The tests were also distributed over different groups in the organization according to the test levels. Unit and integration tests were performed by Software Development Team, and then System Development Team executed system tests. Finally, user acceptance tests and regression tests are performed by the operation team of the bank. However, the organizational structure has changed recently. Today, in a product oriented organization all processes (Requirements Analysis, Design, Realization and Testing) are in the responsibility of the LCMS Application Development Team.

5.2.2. Domain Engineering for LCMS

The domain engineering of LCMS consists of domain objects and domain services. Domain objects are essential core assets, whose instances are the basic structures in campaign and service management. Domain services are the other type of reusable software core assets which are used in construction of all products. Domain objects and services are used for managing the variability and commonality among different products of LCMS. They support building products with lower costs and reduced time to market by enforcing high level of reusability.

The domain objects, whose instances compose the infrastructure of the platform, are:

- ***Reward Pools:*** Database objects that store all the store information about the rewards like cash back or miles.
- ***Target Lists:*** Database objects that hold static or dynamic lists of cards or customers which are included in or excluded from specific campaigns or services.
- ***Campaign Counters:*** Database objects that hold the count or cumulative amount of transactions performed for specific campaigns.
- ***Ledger Records:*** Database objects that hold the monetary records created by campaigns or services for general ledger.
- ***Transaction Logs:*** Database objects that hold the detailed information records for transactions performing insert or update operations on the database.
- ***Reports:*** Printable objects that contain both detailed and summarized information about transactions which are produced automatically or on demand for audit or monetary agreement purposes.

The domain services, which are the basic building blocks for products of LCMS, are:

- ***Campaign Entrance Control:*** Checks if a transaction will run a specific campaign in terms of lower and upper transaction amount limits of the campaign, start and end dates of the campaign and the days of week on which the campaign is supposed to be active.
- ***Target Group Decision:*** Decides whether the cardholder making the transaction is in the target group of the campaign and return the reward multiplier regarding following parameters: Customer and Credit Card Segment, Credit Card Type, Transaction Type, Target List Definition, Date of Member Since, Card Ownership (Primary/Additional)

- **Reward Pool Update:** Updates the reward pool balance with the reward calculated by a campaign mechanism.
- **Campaign Counter Update:** Updates the counters of campaigns which need to count the transaction numbers or amount.
- **Ledger Record Creation:** Creates records for general ledger in case of monetary outcomes while running campaigns.
- **Transaction Log Creation:** Creates log records for all transactions which perform an update or insert operation on the database.

5.2.3. Application Engineering for LCMS

Application engineering of LCMS deals with building products using domain services as reusable components and domain objects for application infrastructure. The products of LCMS are categorized as campaigns and application services. The campaigns and application services are all taking the advantage of reusability on the core assets; however, they all have different product-specific workflows and business logic. These application-specific assets are also handled in application engineering process of LCMS.

There are several campaigns in the scope of LCMS, each one having different reward calculation mechanisms:

- **Cash Back Campaigns:** Cash back reward is calculated for credit or debit card transactions with respect to the transaction type or the type of the purchased goods.
- **Recency, Frequency, Monetary Value (RFM) Campaigns:** Cash back reward is calculated for credit or debit card transactions with respect to the recency or frequency of visits to a specific merchant, the number of purchased goods or the cumulative monetary amount paid in a specific merchant.
- **Installment Campaigns:** Additional installment is calculated for credit card transactions with respect to the transaction amount.
- **Irregular Campaigns:** Regardless of the transactions performed, rewards which are calculated outside of LCMS are charged to reward balances of customers.
- **Central Campaigns:** Any type of rewards can be calculated centrally, independent from channel or transaction type. Transaction/Turnover Counting, Host Generated Reward, Progressive Acquisition, Targeted Acquisition, Reward for Card Issuing Channel and Central Installment can be listed as different sub-types of central campaigns each having

different reward calculation rule sets. Central campaigns are the most complicated products of LCMS which can be run on all channels for any type of transactions.

- ***Discount Campaigns:*** Total or partial amount of the purchase is given back to the customer as a discount in predefined time periods with a special randomizing algorithm.
- ***Postponement Campaigns:*** The payback of a credit card sale is postponed with respect to the several parameters like POS type, transaction type or amount.
- ***Surprise Packages:*** The amount and count of rewards are defined initially as reward packages, and distributed randomly to customers after performing credit card transactions.
- ***Banking Campaigns:*** Rewards can not only be calculated for credit card transactions, but also for a set of banking operations like currency exchange or buying/selling investment funds, etc. with respect to the type and amount of the operation.

In addition to campaigns, there are other products called application services which are mainly dealing with the governance of the whole system:

- ***Advance Services:*** On a commitment accepted by the customer, extra reward can be earned as an advance. The commitment of the customer may either be performing a certain amount of monthly turnover with the credit card, or closing the advance by winning rewards in a predefined period. The system controls the conditions of the commitment periodically and if there is a failure, a monetary penalty is charged.
- ***Ticket Sale Services:*** Customers can buy plane or bus tickets paying with the rewards like cash back or miles that they can earn from campaigns or advance services.
- ***Conversion Services:*** Balances of different reward pools can be converted to each other with respect to some defined rules and multipliers.
- ***Transfer Services:*** Balances can be transferred between the same types of reward pools.
- ***Sign-Up Services:*** Customers can sign up for the target list of a certain campaign through several channels like SMS, Internet or Call Center.
- ***Audit & Correction Services:*** There are special services that can be used only by some authorized users for audit and correction purposes. Viewing the transaction log records or updating the reward pool balances are examples for these services.
- ***Campaign Assignment Services:*** The campaigns are assigned to distribution channels (POS, Virtual POS, Visa, etc) and/or merchant grouping entities (merchant category, group, chain, terminal, etc).

Table 5.2 presents the usage of all campaigns and application services from different distribution channels.

Table 5.2: Usage of LCMS products on distribution channels

Campaigns & Application Services	LCMS Web	POS	Vir. POS	BKM	VISA	MC	CMS	Switch	Call Center	SMS	Branch	Internet
Cash Back Campaigns	✓	✓	✓							✓		
RFM Campaigns	✓	✓	✓							✓		
Instalment Campaigns	✓	✓	✓	✓	✓	✓				✓		
Irregular Campaigns	✓						✓					
Central Campaigns - Transaction/Turnover Counting	✓	✓	✓	✓	✓	✓	✓			✓		
Central Campaigns - Host Generated Rewards	✓			✓	✓	✓	✓			✓		
Central Campaigns - Progressive Acquisition	✓	✓	✓	✓	✓	✓	✓			✓		
Central Campaigns - Targeted Acquisition	✓						✓			✓		
Central Campaigns - Reward for Card Issuing Channel	✓	✓	✓	✓	✓	✓	✓			✓		
Central Campaigns - Central Instalment	✓			✓	✓	✓	✓			✓		
Discount Campaigns	✓	✓	✓									
Postponement Campaigns	✓	✓	✓	✓	✓	✓				✓		
Surprise Packages	✓							✓				
Banking Campaigns	✓								✓		✓	✓
Advance Services	✓								✓			
Ticket Sale Services	✓		✓						✓			
Conversion Services	✓						✓		✓			
Transfer Services	✓						✓		✓			
Sign-Up Services	✓									✓		
Audit and Correction Services	✓								✓			
Campaign Assignment Services	✓											

5.2.4. Commonality and Variability in LCMS

All products in scope of LCMS, i.e. campaigns and application services, have some common procedures and business flows. This commonality is managed by the reusable core assets. For example, every campaign performs a check in the entrance if the cardholder who performed the transaction running this campaign is in the target list. Besides, all the campaigns create a transaction log and most of them create records for general ledger if there is a monetary outcome. All these commonalities are handled in terms of reusability as the basic principle of software product line engineering. Although all campaigns and services share something in common, internal and external variabilities result in the diversity of products. The variabilities among products are handled in two ways, first by reconfigurable and parametric assets in domain engineering artifacts, and then by product-specific constraints in application engineering. Table 5.3 presents some examples of external variabilities and Table 5.4 presents some product constraints on campaign entrance controls as internal variabilities managed in LCMS.

Table 5.3: Examples of external variability in LCMS

Variation Point	Binding Time	Variants
bankCode	Campaign Execution	localBank
		otherBank
poolType	Campaign Definition	Common Cash Back (CCB)
		Private Cash Back (PCB)
		Cross Cash Back (XCB)
		Miles
		Lottery
amountControlType	Campaign Definition	transactionAmount
		provisionAmount
counterLevel	Campaign Definition	customerLevel
		cardLevel
		campaignLevel
counterType	Campaign Execution	RFM_local
		RFM_other
		central
dateControlType	Campaign Definition	dateControl
		timeControl
dayOfWeekControl	Campaign Definition	allDaysOfWeek
		someDaysOfWeek

Table 5.4: Examples of internal variability in LCMS

Products	Internal Variabilities (Product Constraints)
Cash Back Campaigns	Cash Back Campaigns can update CCB/XCB/PCB pool balance depending on the campaign definition.
RFM Campaigns	RFM Campaigns can update CCB/XCB/PCB pool balance depending on the campaign definition.
Installment Campaigns	Installment Campaigns do not update any pool balance.
Irregular Campaigns	Irregular Campaigns can update CCB/XCB/PCB/Miles pool balance depending on the campaign definition.
Central Campaigns	Central Campaigns can update CCB/XCB/PCB/Miles pool balance depending on the campaign definition.
Discount Campaigns	Discount Campaigns do not update any pool balance.

5.3. Problem Definition: Overhead in Regression Testing

The products and services of LCMS highly take advantage of reusability; however, most of the time, this high level of reusability causes difficulties in defining the regression test scope after a new product is derived or existing products are enhanced by new features. When new requirements bring out changes in core assets, every existing product using these core assets needs to be tested whether it works correctly and properly after the changes and reconfiguration in its components. Testing almost all products, as if they are developed from scratch, results in huge regression testing effort and makes the advantage of reusability inoperative. In order to solve this problem, this study addresses a formal modeling of domain core assets which will explicitly show the interdependencies and relations between products, core assets and variation points. The proposed test oriented service and object model is expected to help in determining the necessary regression test scope, thus reducing the testing effort, after adapting the changes in core assets due to the development of a new product or enhancement of existing products in LCMS.

CHAPTER 6

TEST ORIENTED SERVICE AND OBJECT MODEL

Throughout the incremental development of products in a software product line, it is hard to determine the effect of changes in core assets to existing products that are already using those assets. The test oriented model which will be proposed in this chapter aims to clearly present the common structure of the products in application engineering, and the detailed specifications of core assets in domain engineering including their relations and dependencies to the products and variants.

The test oriented service and object model should be applicable to all products in a software product line. This includes both the modeling the general structure of products in application engineering and the reusable core assets in domain engineering, and the relations between them.

6.1. Application Engineering – Product Flow Model

In application engineering, the implementation is based on binding the variants of core assets in order to reconfigure them for the new product, and product-specific features that distinguish a product from the others. For test oriented modeling, the first step, which is performed in application engineering, is constructing a *product flow model* presenting the common structure for similar products reflecting the combination of core assets and product-specific features. Besides, the product flow model should include the execution flow in order to support the inclusion of late bindings which might occur in the runtime.

The high level representation of the products in the flow model should be applicable to all products in the product family. The product flow model, explicitly defining the internal and external variabilities for each product; therefore, should reflect the overall structure, the

execution flow and both optional and mandatory features of the products in the scope of a software product line containing the following items:

- Domain services and related domain objects used by the product
- Application-specific assets and features
- Decision points in the workflow of the products
- Product constraints as internal variabilities
- Mandatory and optional flow blocks or features
- Explicit entry and exit points for the execution of the product

The product flow model will be a combination of figures coming from conventional flow diagrams and representations of variability in feature models or orthogonal variability model. Figure 6.1 shows the graphical notation of the product flow model for application engineering.



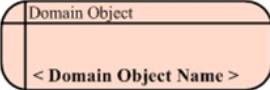
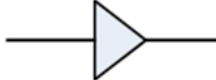
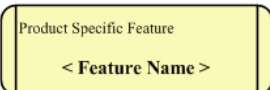




	Entry / Exit points for the execution of a product
	Domain services used in a product
	Domain objects used in or related to a product
	Relations between domain services and objects
	Application specific assets or fetaures
	Optional flow blocks for the execution of a product
	Decision points in the workflow of a product
	Mandatory assets or flows for a product
	Optional assets or flows for a product

Figure 6.1: Graphical notation for product flow model

6.2. Domain Engineering – Service and Object Model

After the construction of the high level product flow model in application engineering, test oriented service and object model for software product lines goes into deeper detail of the core assets in domain engineering. The model includes specifications for both domain objects and services; sub-service decomposition of domain services; and finally test orientation comes with the dependency and traceability matrices produced for all domain services based on product bindings and sub-services.

6.2.1. Specifications for Domain Object Modeling

Domain objects are the conceptual core assets whose instances build up the infrastructure of a software product line. In the test oriented service and object model, domain objects are usually referenced with their relations to domain services and application products. In the model, specifications for each domain object should include the following information:

Domain Object Description: The definition and properties of the domain object including its role and objectives should be explicitly stated.

Related Domain Services: Domain objects are usually related to domain services. Domain services directly operating on the instances of a certain domain object are usually affected from changes on that domain object. This implies the importance of determining the relations between domain objects and services.

Related Products: Some products may have direct dependency on domain objects. The model should associate such products with the domain objects since they probably will be in scope of the regression test when there is a change in the related domain object.

Domain Object Instances: The realization of domain objects by creating instances can be done in several ways. Databases, files, printable reports or any other actual entities can be instances of domain objects. Creating a new instance or modifying an existing instance of a domain object obviously affects the related domain services and application products; therefore, they absolutely have to be considered when defining the test scope.

In the model, the specifications for domain objects including all the necessary test oriented relation and dependency information should be managed in a standard way. The template shown in Table 6.1 should be used for the standardized documentation of domain object specification.

Table 6.1: Document template for domain object specification

CORE ASSET NAME	CORE ASSET TYPE
<DomainObjectName>	Domain Object
CORE ASSET DESCRIPTION	
<Description and properties of the domain object>	
RELATED DOMAIN SERVICES	
<Domain_Service_1>	
...	
<Domain_Service_n>	
RELATED PRODUCTS	
<Application_Product_1>	<Dependency between the domain object and Application_Product_1>
...	...
<Application_Product_n>	<Dependency between the domain object and Application_Product_n>
DOMAIN OBJECT INSTANCES	
<Domain_Object_Instance_1>	<Definition and properties of Domain_Object_Instance_1>
...	...
<Domain_Object_Instance_n>	<Definition and properties of Domain_Object_Instance_n>

6.2.2. Specifications for Domain Service Modeling

Domain services are the software core assets which are highly reusable in the construction of products in a software product line. The proposed model first requires well definition of domain services according to a standard description format. This description should include the essential information about the domain service such as request and response parameters and variabilities handled by that service. Domain services are composed of sub-services. Internal and external variabilities specified in the requirements of a product determine the configuration of a domain service, i.e. combination of sub-services that will be used in that domain service. Test orientation of the model come out with this sub-service decomposition of domain services when the traceability of sub-services and products based on variants are explicitly stated in dependency matrices. For the completion of the test oriented service and object model for software product lines, the following steps should be proceeded for each domain service:

Domain Service Description: The description should include all the essential information about the functionality, input/output parameters, and internal/external variabilities of a domain service. Modeling the domain services starts with the documentation of service descriptions that should be prepared in a standard format presented in Table 6.2.

Table 6.2: Document template for domain service description

CORE ASSET NAME	CORE ASSET TYPE	
<DomainServiceName>	Domain Service	
CORE ASSET DESCRIPTION		
<Description and functionality of the domain service>		
REQUEST PARAMATERS		
<requestParameter_1>	<Definition for requestParameter_1>	
...		
<requestParameter_n>	<Definition for requestParameter_n>	
RESPONSE PARAMATERS		
<responseParameter_1>	<Definition for responseParameter_1>	
...		
<responseParameter_n>	<Definition for responseParameter_n>	
VARIATION POINTS AND VARIANTS (External Variabilities)		
Variation Point	Source	Variants
<VP_1>	<Source of VP_1>	<Variant_a for VP_1>
		<Variant_b for VP_1>
		...
<VP_2>	<Source of VP_2>	<Variant_x for VP_2>
		<Variant_y for VP_2>
		...
PRODUCTS USING THE CORE ASSET		
Products	Constraints (Internal Variabilities)	
<Product_1>	<Constraints as internal variabilities for Product_1>	
...		
<Product_n>	<Constraints as internal variabilities for Product_n>	

Sub-Service Decomposition of Domain Services: A sub-service is a single logical unit of software which performs a specific task and has a special role in the functionality of a domain service. The composition of related sub-services constitutes a domain service. In product construction, the selected domain services are reconfigured in order to use not all but the necessary sub-services based on variants. For each domain service, the sub-service decomposition should be documented using the template shown in Table 6.3.

Table 6.3: Document template for sub-service decomposition

SUB-SERVICE DECOMPOSITION	
Sub-Service Name	Sub-Service Definition
<i><sub_service_1></i>	<i><Definition for functionality of sub_service_1></i>
...	
<i><sub_service_n></i>	<i><Definition for functionality of sub_service_n></i>

Sub-Service Dependencies on Variants: For each domain service, the affinity of sub-services with the probable variants for all variation points should be shown in a dependency matrix given in Table 6.4. For each sub-service, the rows of the matrix should either be filled with a check sign (✓) or a cross sign (X), indicating that the sub-service is used or not used, respectively, for the variants placed in the corresponding columns of the matrix. In other words, if there is a check sign (✓) in the intersection of a sub-service and variant, it means that variant requires the execution of that sub-service and if there is a cross sign (X), it means that variant excludes the execution of that sub-service. If a sub-service has no dependency on a variant, their intersection in the matrix can have the value Not Applicable (N/A) indicating the irrelevancy between them.

Table 6.4: Matrix template for sub-service dependencies on variants

SUB-SERVICE DEPENDENCIES ON VARIANTS			
SUB-SERVICES	VARIATION POINT: <VP_1>		
	<Variant_a for VP_1>	<Variant_b for VP_1>	...
<sub_service_1>			
...			
<sub_service_n>			
SUB-SERVICES	VARIATION POINT: <VP_2>		
	<Variant_x for VP_2>	<Variant_y for VP_2>	...
<sub_service_1>			
...			
<sub_service_n>			

Product Bindings on Variants: The next step is showing the possible bindings of the products on every variation point by preparing another matrix as in the template given in Table 6.5. For each product, the rows of the matrix should either be filled with a check sign (✓) or a cross sign (X), indicating that the product can or can not bind the variant placed in the corresponding column of the matrix. If a variant is irrelevant for a product, their intersection in the matrix can have the value Not Applicable (N/A).

Table 6.5: Matrix template for product bindings on variants

PRODUCT BINDINGS ON VARIANTS			
PRODUCTS	VARIATION POINT: <VP_1>		
	<Variant_a for VP_1>	<Variant_b for VP_1>	...
<Product_1>			
...			
<Product_n>			
PRODUCTS	VARIATION POINT: <VP_2>		
	<Variant_x for VP_2>	<Variant_y for VP_2>	...
<Product_1>			
...			
<Product_n>			

Variant Based Product/Sub-Service Traceability: With the help of the previous two complementary matrices indicating the sub-service dependencies and product bindings on variants, the relations of products and sub-services can easily be consolidated in a single traceability matrix as shown in Table 6.6. The cells in the matrix can again include a check sign (\checkmark), a cross sign (X) or not applicable (N/A). A check sign (\checkmark) means that a product executes a certain sub-service when a specific variant is bound. A cross sign (X) means that the sub-service is not executed for a product when a specific variant is bound. Not applicable (N/A) states that the sub-service and the variant are irrelevant for that product.

This traceability contributes in defining the test scope after the adaption of a new product to the model. When changes are done in domain services due to the requirements of the new product, i.e. a new sub-service is added or an existing sub-service is changed, the impact of this change is limited to only the products which have a check sign for that changed sub-service in a certain row.

Table 6.6: Matrix template for variant based product/sub-service traceability

VARIANT BASED PRODUCT/SUB-SERVICE TRACEABILITY					
VARIANTS		SUB-SERVICES	PRODUCTS		
			<Product_1>	...	<Product_n>
VARIATION POINT: <VP_1>	<Variant_a for VP_1>	<sub_service_1>			
		...			
		<sub_service_n>			
	<Variant_b for VP_1>	<sub_service_1>			
		...			
		<sub_service_n>			
	...	<sub_service_1>			
		...			
		<sub_service_n>			
VARIATION POINT: <VP_2>	<Variant_x for VP_2>	<sub_service_1>			
		...			
		<sub_service_n>			
	<Variant_y for VP_2>	<sub_service_1>			
		...			
		<sub_service_n>			
	...	<sub_service_1>			
		...			
		<sub_service_n>			

6.3. Test Orientation of the Model

The proposed model aims to help defining regression test scope after extensions for new products or enhancements to existing products. Test orientation of the model is primarily based on the following properties:

- 1) Similar to the advantages of service oriented architecture, the decomposition of domain services into loosely coupled sub-services provides certain distinction of functionality.
- 2) The relations and interactions of independent sub-services to both products and domain core assets are explicitly defined.
- 3) The effect of selected variants to domain services and products are clearly designated on sub-service level with the traceability matrices.

After extensions to a software product line, the changes in the variant based product/sub-service traceability matrices can be used in regression test scope determination as follows:

- 1) A new product is added as a column: If there is no change in the existing rows of the traceability matrix, then the new product uses all domain services as they are and has no effect on other products.
- 2) A new sub-service is changed or added as a row: If there is a check sign in a row for that sub-service indicating that it is used by a certain existing product with some specific variant, then that product should be in regression test scope.
- 3) A new variation point or variant is changed or added as multiple rows for sub-services: Similar to addition of a new row for a new sub-service, if there is a check sign in one of the new rows for an existing product, then this product is obviously affected from the changes and should be considered in the regression test.

CHAPTER 7

CASE STUDY: MODELING LCMS

7.1. Scope

The case study on the application of the proposed test oriented service and object model for software product lines includes the following steps:

- 1) The model is adapted on a predefined scope of products in LCMS, including the extensions for latest requirements which are denoted by a different color (blue) for clarity. These extensions include both construction of a new product and changes in existing products.
- 2) The extensions in the model provide a basis for determining what is affected after the realization of new requirements. The coverage of regression test scope after extensions is determined and proved to be complete with the help of the principles of the applied model.

7.2. Implementation of the Proposed Model on LCMS

LCMS, as a software product line, consists of two categories of products, campaigns and application services, which are briefly described in Chapter 5. The application services mainly have the functionality of maintaining and supporting the system, and there were only a few changes in application services after they had been developed. The problem in testing is usually faced when a new campaign is to be added into the scope or when new functionality is assigned to existing campaigns. Therefore, for overcoming the increasing time and effort in testing when new campaigns are developed or when enhancements are done on existing campaigns, the test oriented service and object model is applied to the portion of the system including campaigns only.

The campaigns on which the model applied was selected as the ones initially developed and having a common structure constructed with effective reuse of core assets. First the product flow model is generated and then the core assets specifications were applied to the following campaigns of LCMS, which were constructed sequentially in time:

- Cash Back Campaigns
- RFM Campaigns
- Installment Campaigns
- Irregular Campaigns
- Central Campaigns
- Discount Campaigns

7.2.1. Product Flow Model for LCMS

The product flow model reflects the common structure and the execution flow applicable for the selected campaigns of LCMS. Each campaign starts with entrance controls and target group decision for ensuring the execution. Then, reward is calculated in a campaign specific method and optional steps are followed like updating the reward pool or counters and creating records for general ledger. Finally each campaign execution ends with creating transaction log records. The relation between domain objects and domain services is also depicted in the product flow model of LCMS campaigns. The model includes the decision points for both internal variabilities bound by the constraints and specifications of the campaign and external variabilities especially bound in the runtime. The complete product flow model for the selected campaigns of LCMS is shown in Figure 7.1.

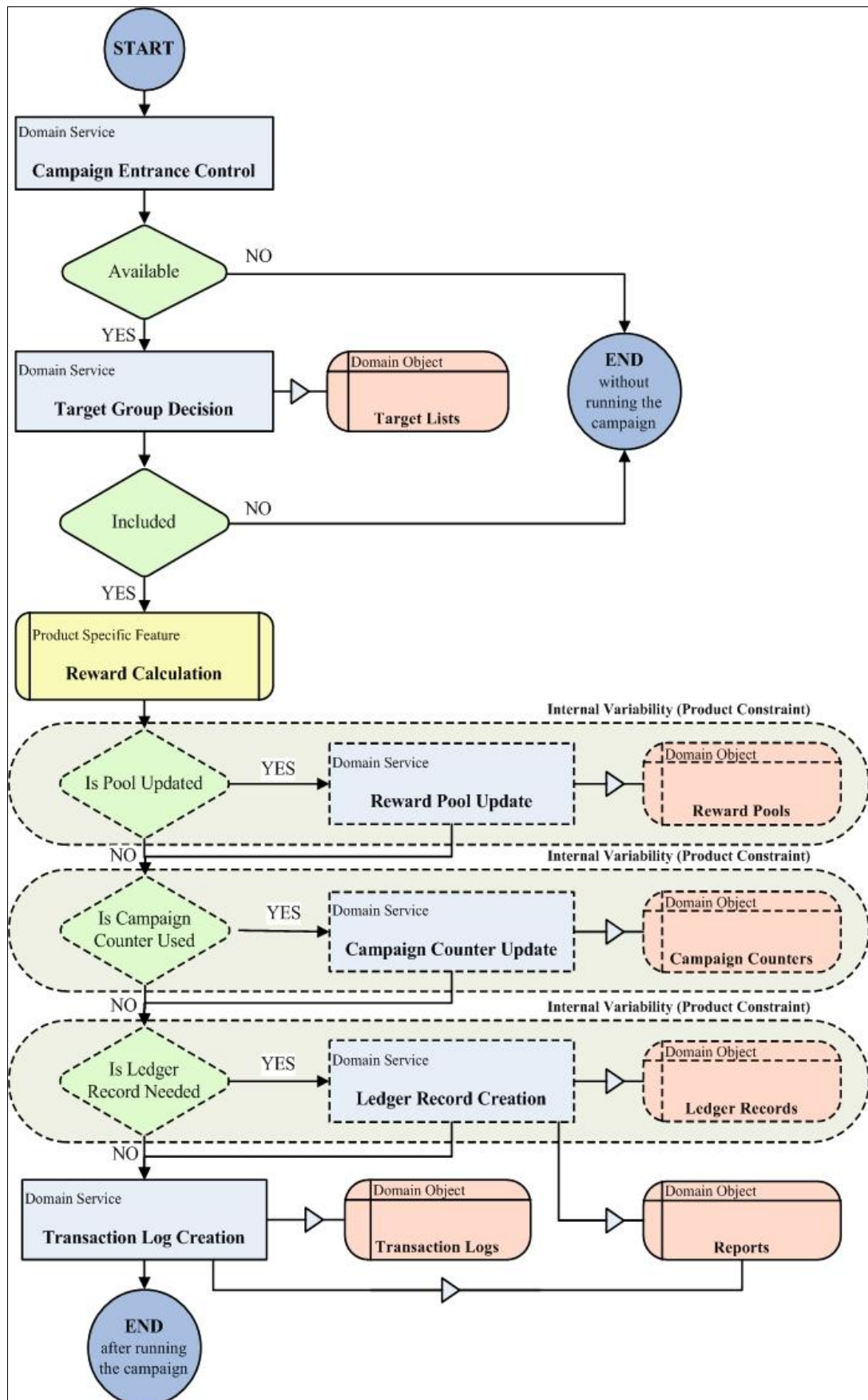


Figure 7.1: Product flow model for LCMS

7.2.2. Modeling Domain Objects of LCMS

For each domain object in the context of LCMS, the specifications are documented conveniently with the templates required for the model. The specification of Reward Pools, one of the major domain objects in LCMS, is presented in Table 7.1.

Table 7.1: Domain object specifications for Reward Pools

CORE ASSET NAME	CORE ASSET TYPE
<i>Reward Pools</i>	Domain Object
CORE ASSET DESCRIPTION	
<i>Database objects that store all the store information about the rewards like cash back or miles.</i>	
RELATED DOMAIN SERVICES	
<i>rewardPoolUpdate</i>	
RELATED PRODUCTS	
<i>Cash Back Campaigns</i>	<i>Cash Back Campaigns directly update the cash back (CCB/PCB/XCB) reward pools.</i>
<i>RFM Campaigns</i>	<i>RFM Campaigns directly update the cash back (CCB/PCB/XCB) and lottery reward pools.</i>
<i>Irregular Campaigns</i>	<i>Irregular Campaigns directly update all the reward pools (cash back/miles).</i>
<i>Central Campaigns</i>	<i>Central Campaigns directly update all the reward pools (cash back/miles/lottery).</i>
DOMAIN OBJECT INSTANCES	
<i>Cash Back Pool</i>	<i>A database table as an instance of Reward Pools for cash back on a basis of card numbers.</i>
<i>Miles Pool</i>	<i>A database table as an instance of Reward Pools for miles on a basis of customer ID numbers.</i>
<i>Lottery Pool</i>	<i>A database table as an instance of Reward Pools for lottery on a basis of card numbers.</i>

The specifications for the rest of the domain objects, i.e. Target Lists, Campaign Counters, Ledger Records, Transaction Logs and Reports, are given in Appendix A.

7.2.3. Modeling Domain Services of LCMS

All domain services of LCMS are documented and related dependency and traceability matrices for each domain service are prepared according to the principles of the test oriented service and object model. The specifications for Reward Pool Update are presented in this section and the specifications for the remaining domain services can be seen in Appendix B.

Table 7.2 shows the domain service description for Reward Pool Update including the primary definitions about the functionality of the service. Other additional information including request and response parameters of the service and related internal and external variation points with their origination sources are also explained in the description document.

Table 7.2: Domain service description for Reward Pool Update

CORE ASSET NAME	CORE ASSET TYPE	
rewardPoolUpdate	Domain Service	
CORE ASSET DESCRIPTION		
Updates the reward pool balance with the reward calculated by a campaign mechanism.		
REQUEST PARAMATERS		
poolID	Unique identification number of the reward pool	
poolType	Type of the pool to be updated	
cardNo	Credit Card number performing the transaction	
customerID	Customer ID of the cardholder performing the transaction	
bankCode	Bank code of the credit card (localBank or otherBank)	
RESPONSE PARAMATERS		
responseCode	1:Pool update successful, 0:Pool update failure	
reasonCode	Reason code if pool update failed	
finalPoolBalance	Final balance of the pool after update	
VARIATION POINTS AND VARIANTS (External Variabilities)		
Variation Point	Source	Variants
poolType	Campaign Definition	CCB
		PCB
		XCB
		Miles
		Lottery
PRODUCTS USING THE CORE ASSET		
Products	Constraints (Internal Variabilities)	
Cash Back Campaigns	Cash Back Campaigns can update CCB/XCB/PCB pool balance depending on the campaign definition.	
RFM Campaigns	RFM Campaigns can update CCB/XCB/PCB pool balance depending on the campaign definition.	
Installment Campaigns	Installment Campaigns do not update any pool balance.	
Irregular Campaigns	Irregular Campaigns can update CCB/XCB/PCB/Miles pool balance depending on the campaign definition.	
Central Campaigns	Central Campaigns can update CCB/XCB/PCB/Miles pool balance depending on the campaign definition.	
Discount Campaigns	Discount Campaigns do not update any pool balance.	

The most important key point in the test orientation of the proposed model is the sub-service decomposition of domain services. Each sub-service can be considered as a single unit of work which has a special role in the functionality of the domain service. The decomposition of the services into indivisible functional units provides and ensures the independency that minimizes the effect of changes. Table 7.3 presents the sub-services which comprise the domain service Reward Pool Update.

Table 7.3: Sub-service decomposition for Reward Pool Update

SUB-SERVICE DECOMPOSITION	
Sub-Service Name	Sub-Service Definition
<i>updatePoolCashBack</i>	<i>Updates the cash back pool balance with the reward calculated by a campaign mechanism.</i>
<i>updatePoolMiles</i>	<i>Updates the miles pool balance with the reward calculated by a campaign mechanism.</i>
<i>updatePoolLottery</i>	<i>Updates the Lottery pool balance with the reward calculated by a campaign mechanism.</i>

The next step for the domain service specification in the model is defining the sub-service dependencies on variants. Each sub-service should be marked if it is used or not when a specific variant is bound. Table 7.4 shows the sub-service dependencies on variants for domain service Reward Pool Update.

Table 7.4: Sub-service dependencies for Reward Pool Update

SUB-SERVICE DEPENDENCIES ON VARIANTS					
SUB-SERVICES	VARIATION POINT: <i>poolType</i>				
	<i>CCB</i>	<i>PCB</i>	<i>XCB</i>	<i>Miles</i>	<i>Lottery</i>
<i>updatePoolCashBack</i>	✓	✓	✓	✗	✗
<i>updatePoolMiles</i>	✗	✗	✗	✓	✗
<i>updatePoolLottery</i>	✗	✗	✗	✗	✓

After the relations of sub-services and variation points, the next matrix presents if variants are bound by the products. The product bindings on variants of the domain service Reward Pool Update is presented in Table 7.5.

Table 7.5: Product bindings for Reward Pool Update

PRODUCT BINDINGS ON VARIANTS					
PRODUCTS	VARIATION POINT: <i>poolType</i>				
	<i>CCB</i>	<i>PCB</i>	<i>XCB</i>	<i>Miles</i>	<i>Lottery</i>
<i>Cash Back Campaigns</i>	✓	✓	✓	✗	✗
<i>RFM Campaigns</i>	✓	✓	✓	✗	✓
<i>Installment Campaigns</i>	✗	✗	✗	✗	✗
<i>Irregular Campaigns</i>	✓	✓	✓	✓	✗
<i>Central Campaigns</i>	✓	✓	✓	✓	✓
<i>Discount Campaigns</i>	✗	✗	✗	✗	✗

The final step in modeling a domain service is the construction of the “product and sub-service traceability matrix” based on variants. This matrix is the final mark-up of the relations and dependencies between products, sub-services, and variants, which will be the key for defining the test scope after changes in the domain services. Table 7.6 shows the functional variability matrix for products on variants for domain service Reward Pool Update.

Table 7.6: Traceability matrix for Reward Pool Update

VARIANT BASED PRODUCT/SUB-SERVICE TRACEABILITY								
VARIANTS		SUB-SERVICES	PRODUCTS					
			Cash Back Campaigns	RFM Campaigns	Installment Campaigns	Irregular Campaigns	Central Campaigns	Discount Campaigns
VARIATION POINT: poolType	CCB	updatePoolCashBack	✓	✓	N/A	✓	✓	N/A
		updatePoolMiles	X	X	N/A	X	X	N/A
		updatePoolLottery	X	X	N/A	X	X	N/A
	PCB	updatePoolCashBack	✓	✓	N/A	✓	✓	N/A
		updatePoolMiles	X	X	N/A	X	X	N/A
		updatePoolLottery	X	X	N/A	X	X	N/A
	XCB	updatePoolCashBack	✓	✓	N/A	✓	✓	N/A
		updatePoolMiles	X	X	N/A	X	X	N/A
		updatePoolLottery	X	X	N/A	X	X	N/A
	Miles	updatePoolCashBack	X	X	N/A	X	X	N/A
		updatePoolMiles	X	X	N/A	✓	✓	N/A
		updatePoolLottery	X	X	N/A	X	X	N/A
	Lottery	updatePoolCashBack	X	X	N/A	X	X	N/A
		updatePoolMiles	X	X	N/A	X	X	N/A
		updatePoolLottery	X	✓	N/A	X	✓	N/A

7.3. Defining Test Scope after Extensions to LCM

The adaptation of the test oriented service and object model on LCMS is presented in previous sections and appendices. On this final version of the model, the last extensions to LCMS are denoted by a different color (blue) for distinction. These extensions include both building a new product in application engineering and changing the core assets in domain engineering. The case study on the implementation of the proposed model to LCMS is finalized with defining the regression test scope using the proposed model after these extensions are adapted.

7.3.1. Latest Requirements

LCMS was comprised of initially developed Cash Back, RFM, Installment, Irregular and Central Campaigns. The new requirements after these five campaigns were originated by the decision of a new card brand. A new campaign type was supposed to be released in parallel with issuing the new brand from scratch. Besides a new campaign, another expectation with the new card brand was extending the existing campaigns with a new reward type. As an addition to the existing reward pools like cash back and miles, the lottery pool has arisen in the scope of the project as a new reward type. In order to keep the goals for time to market, the deadline for the release of the new card brand was very strict as expected.

7.3.2. Building a New Application Product

The expected new product of LCMS was Discount Campaigns. With a special randomizing algorithm, it was supposed to serve the customers free transactions. The design and development of the new campaign was not a time consuming process with the benefits of software product line engineering methodology. The structure of the new campaign was directly applicable to the product flow model of the existing campaigns. By reconfiguring core assets in terms of binding relevant variants and adding the appropriate sub-services to domain services, the campaign was easily constructed. Discount Campaigns used all the domain services except Reward Pool Update, since the reward of the campaign was just a randomly decided free transaction, so there was no need for a pool for that instant reward.

In addition to combination and reconfiguration of core assets, the product-specific functionality was also created as a new campaign mechanism for Discount Campaigns. The application-specific test artifacts like test plans, test cases and test scripts were generated which have traceability on the analysis and design assets. Some test scripts coming from previous campaigns were reused for unit and integration tests of the new product. The reusability in test artifacts reduced the time and effort spent for application tests for Discount Campaigns as for every new product in a software product line.

7.3.3. Extending Core Assets

The extensions to core assets were done due to two reasons: One is for the new application product Discount Campaigns, and the other is for the Lottery Pool as a new reward type for already existing RFM and Central Campaigns.

7.3.3.1. Changes in Domain Objects

Reward Pools: There is no change in Reward Pools for Discount Campaigns. However, a new instance is created as Lottery Pool for the new reward type added to existing campaigns. This new instance requires changes in closely related domain service *rewardPoolUpdate* and affects the products (RFM and Central Campaigns) that will use the lottery pool.

Target Lists: There is no change in Target Lists for either Discount Campaigns or Lottery Pool. The new Discount Campaigns will reuse this domain object as it is.

Campaign Counters: There is no change in Campaign Counters for Lottery Pool. But the Discount Campaigns require a new instance called Discount Counters. This new instance will change the domain service *campaignCounterUpdate*. But this change will not affect any of the existing products since Discount Counters will be a product-specific instance which will only be used by Discount Campaigns.

Ledger Records: There is no change in Ledger Records due to new requirements. Discount Campaigns will create ledger records, but this does not require any changes in existing domain services or products since the ledger records are handled by parameterization defined for each campaign.

Transaction Logs: There is no change in Transaction Logs due to new requirements. Discount Campaigns will create log records, but this does not require any changes in existing domain services or products since the log records need not to be changed.

Reports: New product-specific reports will be created for Discount Campaigns without affecting the existing ones.

7.3.3.2. Changes in Domain Services

Campaign Entrance Control: There is no change due to the new requirements. Discount Campaigns will use this domain service as it is.

Target Group Decision: There is no change due to the new requirements. Discount Campaigns will use this domain service as it is.

Reward Pool Update: A new sub-service will be added to this domain service in order to update the Lottery Pool. Also a new variant, lottery, is defined for the variation point pool type. The new pool type will be applicable for RFM and Central Campaigns.

Campaign Counter Update: A new sub-service will be added to this domain service in order to serve the Discount Campaigns. In addition, new variants are defined for variation points counter level and counter type. The new campaign requires a campaign level counter and discount as a new counter type. It is clear that all these changes in this domain service are specific to Discount Campaigns and has no effect on the existing products.

Ledger Record Creation: A new sub-service will be added to this domain service in order to create ledger records for the Discount Campaigns.

Transaction Log Creation: There is no change due to the new requirements. Discount Campaigns will use this domain service as it is.

7.3.4. Test Scope after Extensions

The first clues for the regression test scope are arising by the changes in domain object specifications in the model. Reward Pools, Campaign Counters and Reports are the domain objects which seem to be changed due to the new requirements. However, they are abstract objects and they are not sufficient for defining the test scope.

Changes in domain services are more determinative elements of the model for defining the regression test scope after extensions. The traceability matrices for domain services will strongly help in defining the test scope for regression tests.

There is no change in domain services Campaign Entrance Control, Target Group Decision and Transaction Log Creation due to the new requirements. Only the integration of these services for the new product will have to be verified and validated during the application-specific system tests for Discount Campaigns.

The changes in Ledger Record Creation and Campaign Counter Update are also specific to Discount Campaigns. Since the new sub-services are specific to Discount Campaigns and they will not be used by any other product, existing products need not to be tested against these extensions for the new campaign.

Regarding the changes in Reward Pool Update shown in Table 7.6, it is clear that RFM and Central Campaigns with new lottery reward pool will be in regression test scope. However, this new variant and sub-service has no relation with other products.

With the help of sub-service decomposition of domain services and traceability matrices, which are the primary facilities of the proposed test oriented service and object model, it can be stated that the changes in core assets for Discount Campaigns do not affect any other existing products and no regression test is needed. Only the changes for the new reward pool, lottery, requires testing of existing products, but this testing is also limited to not all but only affected products.

As a result of the case study, in addition to the application-specific tests for Discount Campaigns, the regression test scope after extensions is proven to be limited to only RFM and Central Campaigns with lottery reward pool. This is a considerable improvement in efficiency for the whole lifecycle of product development in LCMS, since all the products are not in fact affected from a change in a commonly used domain service.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1. Conclusions

This study proposed a test oriented service and object model for software product lines, and implemented this model on the selected scope of products in LCMS for determining the regression test scope after extensions to both application and domain engineering artifacts. Formal modeling of core assets and products explicitly demonstrates which products should be in the regression test scope since they are affected from the changes in the core assets.

The model is based on the principle of sub-service decomposition of domain services, which relates the variants and products to independent units of the domain services. When the changes in domain services are reduced to sub-services, it can be proved that an existing product is not affected from a change in the domain service if it is not directly dependent to that specific sub-service for any variant.

The benefits of this “Test Oriented Service and Object Model” is obviously realized when the extensions are adapted to the traceability matrices as demonstrated in the case study. During the incremental product generation in the previous phases of LCMS, development time and effort for new campaigns were decreasing as expected due to the software product line engineering approach. On the other hand, since the product line platform was evolving, many changes were required in the core assets which resulted in testing nearly all existing products again and again. However, after the implementation of the test oriented service and object model, the reduced regression test scope was observed and the previous trend of increasing testing efforts was reverted. Table 8.1 presents the efficiency gain in development and testing products of LCMS by the utilization of the model, based on the following assumptions and measurable metrics:

Product Size: The approximate size of a campaign in terms of kilo lines of code (KLOC).

Time for Development: The time spent for a campaign development until the beginning of user acceptance and regression tests.

Efficiency in Development: Calculated by *product size* over *time for development*.

Time for Testing: The time spent for user acceptance and regression tests for a campaign.

Efficiency in Testing: Calculated by *product size* over *time for testing*.

Time to Market: The total time spent for the market release of a campaign.

Efficiency in Time to Market: Calculated by *product size* over *time to market*.

In the efficiency calculation, the time spent for the development and testing activities can be used as a single metric for the total cost of developing and testing a product since the number of people in the team remained constant for every product. The *efficiency in development* is increasing by every campaign as expected in a software product line approach. However, until Discount Campaigns, the *efficiency in testing* and *efficiency in time to market* is almost the same for the five initial campaigns. After the adaptation of the model, the effort for user acceptance and regression tests is significantly reduced for Discount Campaigns and the *efficiency in testing* is considerably improved, indicating the overall success of the product line approach which can be seen in the increase of *efficiency in time to market*.

Table 8.1: Efficiency throughput for the test oriented service and object model

	<i>Before modeling LCMS</i>					<i>After modeling LCMS</i>
	Cash Back Campaigns	RFM Campaigns	Installment Campaigns	Irregular Campaigns	Central Campaigns	Discount Campaigns
Product Size (~KLOC)	3,00	5,00	2,00	1,00	5,00	4,00
Time for Development (months)	2,50	6,00	1,50	0,80	3,00	2,00
Efficiency in Development (complexity/time)	1,20	0,83	1,33	1,25	1,67	2,00
Time for Testing (months)	2,00	4,00	3,00	1,00	5,00	1,50
Efficiency in Testing (complexity/time)	1,50	1,25	0,67	1,00	1,00	2,67
Time to Market (months)	4,50	10,00	4,50	1,80	8,00	3,50
Efficiency in Time To Market (complexity/time)	0,67	0,50	0,44	0,56	0,63	1,14

Figure 8.1 helps for further clearance on the efficiency gain after the implementation of the model. The model was applied to LCMS right before the development of Discount Campaigns. Before using the model, if a domain service was changed due to a new requirement or a new product, all the previous products using that domain service were in the scope of the regression test. Until the adaptation of the model, although the efficiency in development was increasing, the efficiency in testing was in a decreasing. This resulted in only a slight increase in time to market. However, after the adaptation of the model with Discount Campaigns, there was a considerable increase in the efficiency of testing process which also resulted in reduced time to market.

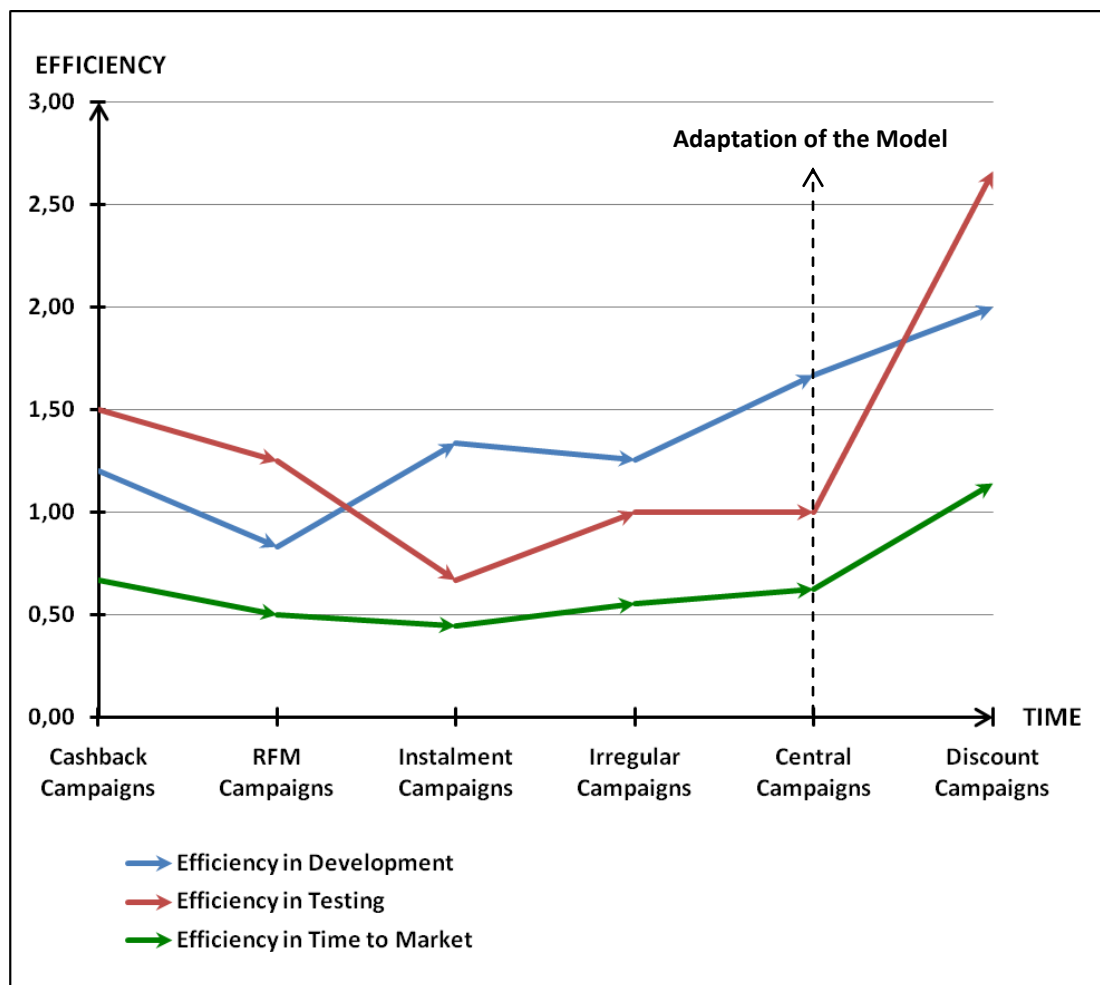


Figure 8.1: Efficiency graph for the test oriented service and object model

8.2. Future Work

The contribution with the test oriented service and object model for software product lines in this thesis covered both application and domain engineering. However, the concentration was mainly on the core assets of the domain engineering with sub-service decomposition and traceability matrices. The product flow model proposed for application engineering environment was only a complementary reinforcement for reflecting the common structure of the products and it was practiced superficially. Further research and improvement on the product flow model, especially for the full integration and representation of variability, can be a subject for future work.

The adaptation of the proposed model on a sample software product line was performed manually as a case study in this thesis. Designing and implementing a modeling tool, which will support the graphical notation for the product flow model, the documentation of core asset specifications and the traceability matrices with a graphical user interface may utilize the adaptation and maintenance of the model.

Finally, the test oriented service and object model can be enriched with integration and management of reusable test artifacts where possible, for further efficiency improvement in testing software product lines.

REFERENCES

- [1] Paul Clements, Linda Northrop. "Software Product Lines: Practices and Patterns". Addison-Wesley Professional. Boston. 2001.
- [2] Klaus Pohl, Günter Böckle, Frank J. van der Linden. "Software Product Line Engineering: Foundations, Principles and Techniques". Springer-Verlag New York, Secaucus, NJ. September 2005.
- [3] Frank J. van der Linden, Klaus Schmid, Eelco Rommes. "Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering". Springer-Verlag New York, Secaucus, NJ. July 2007.
- [4] Ivar Jacobson, Martin Griss, Patrik Jonsson. "Software Reuse: Architecture, Process and Organization for Business Success". ACM Press/Addison-Wesley Publishing Co. 1997.
- [5] Paul Clements, Linda Northrop. "Software Product Lines". Presentation, Software Engineering Institute (SEI), Carnegie Mellon University. 2003.
- [6] Charles W. Krueger. "Introduction to the Emerging Practice of Software Product Line Development". Methods & Tools, Volume 14, Number 3, pages 3-15. Fall 2006.
- [7] Stephen R. Schach, Amir Tomer. "Development/Maintenance/Reuse: Software Evolution in Product Lines". First Conference on Software Product Lines: Experience and Research Directions, pages 437-450, Denver, Colorado, United States. November 2000.
- [8] Charles W. Krueger, William A. Hetrick, Joseph G. Moore. "Making an Incremental Transition to Software Product Line Practice". Methods & Tools, pages 16-27. Fall 2006.
- [9] Jan Bosch. "Organizing for Software Product Lines". 3rd International Workshop on Software Architectures for Product Families (IWSAPF-3). March 2000.
- [10] Jan Bosch. "Maturity and Evolution in Software Product Lines: Approaches, Artifacts and Organization". Second Conference Software Product Line Conference (SPLC2), pp. 257-271. August 2002.
- [11] James Coplien, Daniel Hoffman, David Weiss. "Commonality and Variability in Software Engineering". IEEE Software, pages 37-45. November/December 1999.

- [12] Felix Bachmann, Len Bass. "Managing Variability in Software Architectures". 2001 Symposium on Software Reusability: Putting Software Reuse in Context, pages 126-132, Toronto, Ontario, Canada. May 2001.
- [13] Jilles Van Gorp, Jan Bosch, Mikael Svahnberg. "On the Notion of Variability in Software Product Lines". Working IEEE/IFIP Conference on Software Architecture (WICSA'01), page 45. August 28-31, 2001.
- [14] Michalis Anastasopoulos, Cristina Gacek. "Implementing Product Line Variabilities". SIGSOFT Software Engineering Notes, Volume 26, Issue 3, pages 109-117. May 2001.
- [15] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, Henk Obbink, Klaus Pohl. "Variability Issues in Software Product Lines". 4th International Workshop on Product Family Engineering (PFE-4). October 2001.
- [16] Mikael Svahnberg, Jan Bosch. "Issues Concerning Variability in Software Product Lines". International Workshop on Software Architectures for Product Families, pages 146-157. March 15-17, 2000.
- [17] Jilles van Gorp, Jan Bosch, Mikael Svahnberg. "Managing Variability in Software Product Lines". Landelijk Architectuur Congres, Amsterdam. 2000.
- [18] Henry Muccini, Andre van der Hoek. "Towards Testing Product Line Architectures". Electronic Notes in Theoretical Computer Science, Volume 82, Issue 6, Pages 99-109. September 2003.
- [19] Hui Zeng, Wendy Zhang, David Rine. "Analysis of Testing Effort by Using Core Assets in Software Product Line Testing". International Workshop on Software Product Line Testing (SPLiT 2004). 2004.
- [20] John D. McGregor, Prakash Sodhani, Sai Madhavapeddi. "Testing Variability in a Software Product Line". International Workshop on Software Product Line Testing (SPLiT 2004). 2004.
- [21] Erika Mir Olimpiew, Hassan Gomaa. "Reusable System Tests for Applications Derived from Software Product Lines". International Workshop on Software Product Line Testing (SPLiT 2005). 2005.
- [22] Peter Knauber, William Hetrick. "Product Line Testing and Product Line Development - Variations on a Common Theme". International Workshop on Software Product Line Testing (SPLiT 2005). 2005.
- [23] Leire Etxeberria, Goiuria Sagardui, Lorea Belategi. "Quality Aware Software Product Line Engineering". J. Braz. Comp. Soc., vol.14, no.1, pages 57-69. ISSN 0104-6500. March 2008.
- [24] Jonas Hörnstein, Håkan Edler. "Configuration and Testing of Components in Software Product Lines". Second Conference on Software Engineering Research and Practise in

Sweden (SERPS'02). Blekinge Institute of Technology, Karlskrona, Sweden. October 24-25, 2002.

- [25] John D. McGregor. "Testing a Software Product Line". Technical Report, CMU/SEI-2001-TR-022. December 2001.
- [26] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, Yves Bontemps. "Generic Semantics of Feature Diagrams". Computer Networks 51, pages 456-479. 2007.

APPENDIX A

DOMAIN OBJECT SPECIFICATIONS OF LCMS

A.1. Target Lists

Table A.1: Domain object specifications for Target Lists

CORE ASSET NAME	CORE ASSET TYPE
<i>Target Lists</i>	Domain Object
CORE ASSET DESCRIPTION	
<i>Database objects that hold static or dynamic lists of cards or customers which are included in or excluded from specific campaigns or services.</i>	
RELATED DOMAIN SERVICES	
<i>targetGroupDecision</i>	
RELATED PRODUCTS	
<i>Cash Back Campaigns</i>	<i>A list of customers can be included or excluded from the target group of Cash Back Campaigns.</i>
<i>RFM Campaigns</i>	<i>A list of customers can be included or excluded from the target group of RFM Campaigns.</i>
<i>Installment Campaigns</i>	<i>A list of customers can be included or excluded from the target group of Installment Campaigns.</i>
<i>Central Campaigns</i>	<i>A list of customers can be included or excluded from the target group of Central Campaigns.</i>
<i>Discount Campaigns</i>	<i>A list of customers can be included or excluded from the target group of Discount Campaigns.</i>
DOMAIN OBJECT INSTANCES	
<i>Include Lists</i>	<i>A database table as an instance of Target Lists for including a set of cards/customers in the target group of a campaign.</i>
<i>Exclude Lists</i>	<i>A database table as an instance of Target Lists for excluding a set of cards/customers from the target group of a campaign.</i>

A.2. Campaign Counters

Table A.2: Domain object specifications for Campaign Counters

CORE ASSET NAME	CORE ASSET TYPE
<i>Campaign Counters</i>	Domain Object
CORE ASSET DESCRIPTION	
<i>Database objects that hold the count or cumulative amount of transactions performed for specific campaigns.</i>	
RELATED DOMAIN SERVICES	
<i>campaignCounterUpdate</i>	
RELATED PRODUCTS	
<i>RFM Campaigns</i>	<i>The frequency of visits, the number of purchased goods or the cumulative monetary amount is stored in counters of RFM Campaigns.</i>
<i>Central Campaigns</i>	<i>Cumulative number or monetary amount of transactions are counted and stored in counters of Central Campaigns.</i>
<i>Discount Campaigns</i>	<i>Number of discounts per a specific period is counted and stored in counters of Discount Campaigns.</i>
DOMAIN OBJECT INSTANCES	
<i>RFM Counters</i>	<i>A database table as an instance of Campaign Counters for RFM Campaigns.</i>
<i>Central Counters</i>	<i>A database table as an instance of Campaign Counters for Central Campaigns.</i>
<i>Discount Counters</i>	<i>A database table as an instance of Campaign Counters for Discount Campaigns.</i>

A.3. Ledger Records

Table A.3: Domain object specifications for Ledger Records

CORE ASSET NAME	CORE ASSET TYPE
<i>Ledger Records</i>	Domain Object
CORE ASSET DESCRIPTION	
<i>Database objects that hold the monetary records created by campaigns or services for general ledger.</i>	
RELATED DOMAIN SERVICES	
<i>ledgerRecordCreation</i>	
RELATED PRODUCTS	
<i>Cash Back Campaigns</i>	<i>Cash Back Campaigns create ledger records for monetary operations on cash back pool.</i>
<i>RFM Campaigns</i>	<i>RFM Campaigns create ledger records for monetary operations on cash back pool.</i>
<i>Irregular Campaigns</i>	<i>Irregular Campaigns create ledger records for monetary operations on cash back and miles pool.</i>
<i>Central Campaigns</i>	<i>Central Campaigns create ledger records for monetary operations on cash back and miles pool.</i>
<i>Discount Campaigns</i>	<i>Discount Campaigns create ledger records for monetary operations on discounts.</i>
DOMAIN OBJECT INSTANCES	
<i>Ledger Record Table</i>	<i>A database table as an instance of Ledger Records for monetary operations.</i>

A.4. Transaction Logs

Table A.4: Domain object specifications for Transaction Logs

CORE ASSET NAME	CORE ASSET TYPE
<i>Transaction Logs</i>	Domain Object
CORE ASSET DESCRIPTION	
<i>Database objects that hold the detailed information records for transactions performing insert or update operations on the database.</i>	
RELATED DOMAIN SERVICES	
<i>transactionLogCreation</i>	
RELATED PRODUCTS	
<i>Cash Back Campaigns</i>	<i>Cash Back Campaigns create transaction logs on reward calculation.</i>
<i>RFM Campaigns</i>	<i>RFM Campaigns create transaction logs on reward calculation and counter updates.</i>
<i>Installment Campaigns</i>	<i>Installment Campaigns create transaction logs on reward calculation.</i>
<i>Irregular Campaigns</i>	<i>Irregular Campaigns create transaction logs on reward calculation.</i>
<i>Central Campaigns</i>	<i>Central Campaigns create transaction logs on reward calculation and counter updates.</i>
<i>Discount Campaigns</i>	<i>Discount Campaigns create transaction logs on discount calculation and counter updates.</i>
DOMAIN OBJECT INSTANCES	
<i>Daily Transaction Logs</i>	<i>A database table as an instance of Transaction Logs for daily storage.</i>
<i>Transaction Log History</i>	<i>A database table as an instance of Transaction Logs for historical storage.</i>

A.5. Reports

Table A.5: Domain object specifications for Reports

CORE ASSET NAME	CORE ASSET TYPE
<i>Reports</i>	Domain Object
CORE ASSET DESCRIPTION	
<i>Printable objects that contain both detailed and summarized information about transactions which are produced automatically or on demand for audit or monetary agreement purposes.</i>	
RELATED DOMAIN SERVICES	
<i>transactionLogCreation</i>	
<i>ledgerRecordCreation</i>	
RELATED PRODUCTS	
<i>Cash Back Campaigns</i>	<i>Cash Back Campaigns are included in the reports in terms of transaction logs and ledger records.</i>
<i>RFM Campaigns</i>	<i>RFM Campaigns are included in the reports in terms of transaction logs and ledger records.</i>
<i>Installment Campaigns</i>	<i>Installment Campaigns are included in the reports in terms of transaction logs.</i>
<i>Irregular Campaigns</i>	<i>Irregular Campaigns are included in the reports in terms of transaction logs and ledger records.</i>
<i>Central Campaigns</i>	<i>Central Campaigns are included in the reports in terms of transaction logs and ledger records.</i>
<i>Discount Campaigns</i>	<i>Discount Campaigns are included in the reports in terms of transaction logs and ledger records.</i>
DOMAIN OBJECT INSTANCES	
<i>Daily Monetary Agreement for Cash Back</i>	<i>A printable report for monetary agreement of cash back pool operations produced from ledger records every day.</i>
<i>Daily Monetary Agreement for Miles</i>	<i>A printable report for monetary agreement of miles pool operations produced from ledger records every day.</i>
<i>Monthly Earning/Usage Report for Cash Back</i>	<i>A printable report for earnings and usage of cash back rewards produced from transaction logs every month.</i>
<i>Monthly Earning/Usage Report for Miles</i>	<i>A printable report for earnings and usage of miles rewards produced from transaction logs every month.</i>
<i>Daily Monetary Agreement for Discount</i>	<i>A printable report for monetary agreement of discount operations produced from ledger records every day.</i>
<i>Daily Discount Transaction Report</i>	<i>A printable report for discounts produced from transaction logs every day.</i>

APPENDIX B

DOMAIN SERVICE SPECIFICATIONS OF LCMS

B.1. Campaign Entrance Control

Table B.1: Domain service description for Campaign Entrance Control

CORE ASSET NAME	CORE ASSET TYPE	
campaignEntranceControl	Domain Service	
CORE ASSET DESCRIPTION		
Control if the transaction will run a specific campaign in terms of: * Lower and upper transaction amount limits of the campaign * Start and End dates of the campaign * The days of week on which the campaign is supposed to be active		
REQUEST PARAMATERS		
campaignID	Unique identification number of the campaign	
campaignType	1:Cash Back 2:RFM 3:Installment 4:Irregular 5:Central 6:Discount	
transactionAmount	Gross amount of the transaction including cash back withdrawal	
provisionAmount	Plain amount of the transaction excluding cash back withdrawal	
transactionDateTime	The date and time on which the transaction is performed	
bankCode	Bank code of the credit card (localBank or otherBank)	
RESPONSE PARAMATERS		
responseCode	1:Included in the campaign, 0:Excluded from the campaign	
reasonCode	Reason code if excluded from the campaign	
VARIATION POINTS AND VARIANTS (External Variabilities)		
Variation Point	Source	Variants
amountControlType	Campaign Definition	transactionAmount
		provisionAmount
dateControlType	Campaign Definition	dateControl
		timeControl
dayOfWeekControl	Campaign Definition	allDaysOfWeek
		someDaysOfWeek

Table B.1 (continued): Domain service description for Campaign Entrance Control

PRODUCTS USING THE CORE ASSET	
Products	Constraints (Internal Variabilities)
<i>Cash Back Campaigns</i>	<i>Campaign entrance controls are done based on transaction amount, begin - end dates and campaign always runs on all days of week.</i>
<i>RFM Campaigns</i>	<i>Campaign entrance controls are done based on either transaction or provision amount, begin - end dates and campaign always runs on all days of week.</i>
<i>Installment Campaigns</i>	<i>Campaign entrance controls are done based on either transaction or provision amount, begin - end dates and campaign might run on some days of week.</i>
<i>Irregular Campaigns</i>	<i>There is no amount or date control for irregular campaigns and they can run on all days of week.</i>
<i>Central Campaigns</i>	<i>Campaign entrance controls are done based on either transaction or provision amount, begin - end times and campaign might run on some days of week.</i>
<i>Discount Campaigns</i>	<i>Campaign entrance controls are done based on either transaction or provision amount, begin - end dates and campaign might run on some days of week.</i>

Table B.2: Sub-service decomposition for Campaign Entrance Control

SUB-SERVICE DECOMPOSITION	
Sub-Service Name	Sub-Service Definition
<i>checkTransactionAmount</i>	<i>Check the transaction amount whether it is in between lower and upper limits of the campaign.</i>
<i>checkProvisionAmount</i>	<i>Check the provision amount whether it is in between lower and upper limits of the campaign.</i>
<i>checkStartEndDate</i>	<i>Check the transaction date whether it is in between start and end date of the campaign.</i>
<i>checkStartEndTime</i>	<i>Check the transaction time whether it is in between start and end time of the campaign.</i>
<i>checkDaysOfWeek</i>	<i>Check if the transaction day is one of the days of week on which the campaign is supposed to be run.</i>

Table B.3: Sub-service dependencies for Campaign Entrance Control

SUB-SERVICE DEPENDENCIES ON VARIANTS		
SUB-SERVICES	VARIATION POINT: <i>amountControlType</i>	
	<i>transactionAmount</i>	<i>provisionAmount</i>
<i>checkTransactionAmount</i>	✓	X
<i>checkProvisionAmount</i>	X	✓
<i>checkStartEndDate</i>	N/A	N/A
<i>checkStartEndTime</i>	N/A	N/A
<i>checkDaysOfWeek</i>	N/A	N/A
SUB-SERVICES	VARIATION POINT: <i>dateControlType</i>	
	<i>dateControl</i>	<i>timeControl</i>
<i>checkTransactionAmount</i>	N/A	N/A
<i>checkProvisionAmount</i>	N/A	N/A
<i>checkStartEndDate</i>	✓	X
<i>checkStartEndTime</i>	X	✓
<i>checkDaysOfWeek</i>	N/A	N/A
SUB-SERVICES	VARIATION POINT: <i>dayOfWeekControl</i>	
	<i>allDaysOfWeek</i>	<i>someDaysOfWeek</i>
<i>checkTransactionAmount</i>	N/A	N/A
<i>checkProvisionAmount</i>	N/A	N/A
<i>checkStartEndDate</i>	N/A	N/A
<i>checkStartEndTime</i>	N/A	N/A
<i>checkDaysOfWeek</i>	✓	X

Table B.4: Product bindings for Campaign Entrance Control

PRODUCT BINDINGS ON VARIANTS		
PRODUCTS	VARIATION POINT: <i>amountControlType</i>	
	<i>transactionAmount</i>	<i>provisionAmount</i>
<i>Cash Back Campaigns</i>	✓	X
<i>RFM Campaigns</i>	✓	✓
<i>Installment Campaigns</i>	✓	✓
<i>Irregular Campaigns</i>	X	X
<i>Central Campaigns</i>	✓	✓
<i>Discount Campaigns</i>	✓	✓
PRODUCTS	VARIATION POINT: <i>dateControlType</i>	
	<i>dateControl</i>	<i>timeControl</i>
<i>Cash Back Campaigns</i>	✓	X
<i>RFM Campaigns</i>	✓	X
<i>Installment Campaigns</i>	✓	X
<i>Irregular Campaigns</i>	X	X
<i>Central Campaigns</i>	X	✓
<i>Discount Campaigns</i>	✓	X
PRODUCTS	VARIATION POINT: <i>dayOfWeekControl</i>	
	<i>allDaysOfWeek</i>	<i>someDaysOfWeek</i>
<i>Cash Back Campaigns</i>	✓	X
<i>RFM Campaigns</i>	✓	X
<i>Installment Campaigns</i>	X	✓
<i>Irregular Campaigns</i>	✓	X
<i>Central Campaigns</i>	X	✓
<i>Discount Campaigns</i>	X	✓

Table B.5: Traceability matrix for Campaign Entrance Control

VARIANT BASED PRODUCT/SUB-SERVICE TRACEABILITY								
VARIANTS		SUB-SERVICES	PRODUCTS					
			Cash Back Campaigns	RFM Campaigns	Installment Campaigns	Irregular Campaigns	Central Campaigns	Discount Campaigns
VARIATION POINT: amountControlType	transactionAmount	checkTransactionAmount	✓	✓	✓	✗	✓	✓
		checkProvisionAmount	N/A	N/A	N/A	N/A	N/A	N/A
		checkStartDate	N/A	N/A	N/A	N/A	N/A	N/A
		checkStartTime	N/A	N/A	N/A	N/A	N/A	N/A
		checkDaysOfWeek	N/A	N/A	N/A	N/A	N/A	N/A
	provisionAmount	checkTransactionAmount	N/A	N/A	N/A	N/A	N/A	N/A
		checkProvisionAmount	✗	✓	✓	✗	✓	✓
		checkStartDate	N/A	N/A	N/A	N/A	N/A	N/A
		checkStartTime	N/A	N/A	N/A	N/A	N/A	N/A
		checkDaysOfWeek	N/A	N/A	N/A	N/A	N/A	N/A
VARIATION POINT: dateControlType	dateControl	checkTransactionAmount	N/A	N/A	N/A	N/A	N/A	N/A
		checkProvisionAmount	N/A	N/A	N/A	N/A	N/A	N/A
		checkStartDate	✓	✓	✓	✗	✗	✓
		checkStartTime	N/A	N/A	N/A	N/A	N/A	N/A
		checkDaysOfWeek	N/A	N/A	N/A	N/A	N/A	N/A
	timeControl	checkTransactionAmount	N/A	N/A	N/A	N/A	N/A	N/A
		checkProvisionAmount	N/A	N/A	N/A	N/A	N/A	N/A
		checkStartDate	N/A	N/A	N/A	N/A	N/A	N/A
		checkStartTime	✗	✗	✗	✗	✓	✗
		checkDaysOfWeek	N/A	N/A	N/A	N/A	N/A	N/A
VARIATION POINT: dayOfWeekControl	allDaysOfWeek	checkTransactionAmount	N/A	N/A	N/A	N/A	N/A	N/A
		checkProvisionAmount	N/A	N/A	N/A	N/A	N/A	N/A
		checkStartDate	N/A	N/A	N/A	N/A	N/A	N/A
		checkStartTime	N/A	N/A	N/A	N/A	N/A	N/A
		checkDaysOfWeek	N/A	N/A	N/A	N/A	N/A	N/A
	someDaysOfWeek	checkTransactionAmount	N/A	N/A	N/A	N/A	N/A	N/A
		checkProvisionAmount	N/A	N/A	N/A	N/A	N/A	N/A
		checkStartDate	N/A	N/A	N/A	N/A	N/A	N/A
		checkStartTime	N/A	N/A	N/A	N/A	N/A	N/A
		checkDaysOfWeek	✗	✗	✓	✗	✓	✓

B.2. Target Group Decision

Table B.6: Domain service description for Target Group Decision

CORE ASSET NAME	CORE ASSET TYPE	
targetGroupDecision	Domain Service	
CORE ASSET DESCRIPTION		
Decide whether the cardholder making the transaction is in the target group of the campaign and return the reward multiplier regarding following parameters: <ul style="list-style-type: none">* Customer Segment* Credit Card Segment* Credit Card Logo* Other Bank Card Logo* Transaction Function ID* Target List Definition* Date of Member Since* Card Ownership (Primary/Additional)		
REQUEST PARAMATERS		
campaignID	Unique identification number of the campaign	
campaignType	1:Cash Back 2:RFM 3:Installment 4:Irregular 5:Central 6:Discount	
cardNo	Credit Card number performing the transaction	
customerID	Customer ID of the cardholder performing the transaction	
bankCode	Bank code of the credit card (localBank or otherBank)	
RESPONSE PARAMATERS		
responseCode	1:Included in the target group, 0:Excluded from the target group	
reasonCode	reason code if excluded from the target group	
rewardMultiplier	calculated regarding customer segment, credit card segment, credit card logo or other bank's card logo	
VARIATION POINTS AND VARIANTS (External Variabilities)		
Variation Point	Source	Variants
bankCode	Campaign Execution	localBank
		otherBank
PRODUCTS USING THE CORE ASSET		
Products	Constraints (Internal Variabilities)	
Cash Back Campaigns	All functionality is operative for both local and other banks.	
RFM Campaigns	All functionality is operative for both local and other banks.	
Installment Campaigns	All functionality is operative for both local and other banks except checkFunctionID since this is not a business requirement for Installment Campaigns.	
Irregular Campaigns	Just getCustomerInfo and getCardInfo services are relevant for local bank. Irregular Campaigns are not valid for other banks.	
Central Campaigns	All functionality is operative for both local and other banks except checkFunctionID since this is not a business requirement for Central Campaigns.	
Discount Campaigns	All functionality is operative for local bank except checkFunctionID since this is not a business requirement for discount campaigns. Discount campaigns are not valid for other banks.	

Table B.7: Sub-service decomposition for Target Group Decision

SUB-SERVICE DECOMPOSITION	
Sub-Service Name	Sub-Service Definition
<i>getCustomerInfo</i>	<i>Inquire for customer information from Customer Information File (CIF).</i>
<i>getCardInfo</i>	<i>Inquire for card information from Card Management System (CMS).</i>
<i>checkCustomerSegment</i>	<i>Check whether customer's segment is included in the campaign and get "Customer Segment Reward Multiplier" if included.</i>
<i>checkCreditCardSegment</i>	<i>Check whether credit card's segment is included in the campaign and get "Credit Card Segment Reward Multiplier" if included.</i>
<i>checkLocalBankLogo</i>	<i>Check whether credit card's logo type is included in the campaign and get "Credit Card Logo Reward Multiplier" if included.</i>
<i>checkOtherBankLogo</i>	<i>Check whether other bank credit card's logo type is included in the campaign and get "Other Bank Credit Card Logo Reward Multiplier" if included.</i>
<i>checkFunctionID</i>	<i>Check whether the function ID of the transaction is included in the campaign.</i>
<i>checkTargetList</i>	<i>Check whether a target list is defined for the campaign and if defined, check whether the customer ID or card No is included or excluded.</i>
<i>checkMemberSince</i>	<i>Check whether the membership of the cardholder is older than the campaign limits.</i>
<i>checkCardOwnership</i>	<i>Check whether the credit card is Primary or Additional and check if it is included in the campaign constraints.</i>

Table B.8: Sub-service dependencies for Target Group Decision

SUB-SERVICE DEPENDENCIES ON VARIANTS		
SUB-SERVICES	VARIATION POINT: <i>bankCode</i>	
	<i>localBank</i>	<i>otherBank</i>
<i>getCustomerInfo</i>	✓	✗
<i>getCardInfo</i>	✓	✗
<i>checkCustomerSegment</i>	✓	✗
<i>checkCreditCardSegment</i>	✓	✗
<i>checkLocalBankLogo</i>	✓	✗
<i>checkOtherBankLogo</i>	✗	✓
<i>checkFunctionID</i>	✓	✓
<i>checkTargetList</i>	✓	✗
<i>checkMemberSince</i>	✓	✗
<i>checkCardOwnership</i>	✓	✗

Table B.9: Product bindings for Target Group Decision

PRODUCT BINDINGS ON VARIANTS		
PRODUCTS	VARIATION POINT: <i>bankCode</i>	
	<i>localBank</i>	<i>otherBank</i>
<i>Cash Back Campaigns</i>	✓	✓
<i>RFM Campaigns</i>	✓	✓
<i>Installment Campaigns</i>	✓	✓
<i>Irregular Campaigns</i>	✓	✗
<i>Central Campaigns</i>	✓	✓
<i>Discount Campaigns</i>	✓	✗

Table B.10: Traceability matrix for Target Group Decision

VARIANT BASED PRODUCT/SUB-SERVICE TRACEABILITY								
VARIANTS	SUB-SERVICES	PRODUCTS						
		Cash Back Campaigns	RFM Campaigns	Installment Campaigns	Irregular Campaigns	Central Campaigns	Discount Campaigns	
VARIATION POINT: <i>bankCode</i>	<i>localBank</i>	<i>getCustomerInfo</i>	✓	✓	✓	✓	✓	✓
		<i>getCardInfo</i>	✓	✓	✓	✓	✓	✓
		<i>checkCustomerSegment</i>	✓	✓	✓	✗	✓	✓
		<i>checkCreditCardSegment</i>	✓	✓	✓	✗	✓	✓
		<i>checkLocalBankLogo</i>	✓	✓	✓	✗	✓	✓
		<i>checkOtherBankLogo</i>	✗	✗	✗	✗	✗	✗
		<i>checkFunctionID</i>	✓	✓	✗	✗	✗	✗
		<i>checkTargetList</i>	✓	✓	✓	✗	✓	✓
		<i>checkMemberSince</i>	✓	✓	✓	✗	✓	✓
		<i>checkCardOwnership</i>	✓	✓	✓	✗	✓	✓
	<i>otherBank</i>	<i>getCustomerInfo</i>	✗	✗	✗	✗	✗	✗
		<i>getCardInfo</i>	✗	✗	✗	✗	✗	✗
		<i>checkCustomerSegment</i>	✗	✗	✗	✗	✗	✗
		<i>checkCreditCardSegment</i>	✗	✗	✗	✗	✗	✗
		<i>checkLocalBankLogo</i>	✗	✗	✗	✗	✗	✗
		<i>checkOtherBankLogo</i>	✓	✓	✓	✗	✓	✓
		<i>checkFunctionID</i>	✓	✓	✗	✗	✗	✗
		<i>checkTargetList</i>	✗	✗	✗	✗	✗	✗
		<i>checkMemberSince</i>	✗	✗	✗	✗	✗	✗
		<i>checkCardOwnership</i>	✗	✗	✗	✗	✗	✗

B.3. Campaign Counter Update

Table B.11: Domain service description for Campaign Counter Update

CORE ASSET NAME	CORE ASSET TYPE	
campaignCounterUpdate	Domain Service	
CORE ASSET DESCRIPTION		
Updates the counters of campaigns which need to count the transaction numbers or amount.		
REQUEST PARAMATERS		
counterType	Type of the counter to be updated	
counterLevel	Level of the counter (card no/customer ID)	
cardNo	Credit Card number performing the transaction	
customerID	Customer ID of the cardholder performing the transaction	
bankCode	Bank code of the credit card (localBank or otherBank)	
RESPONSE PARAMATERS		
responseCode	1:Counter update successful, 0:Counter update failure	
reasonCode	Reason code if counter update failed	
finalCounterValues	Final values of the counter after update	
VARIATION POINTS AND VARIANTS (External Variabilities)		
Variation Point	Source	Variants
bankCode	Campaign Execution	localBank
		otherBank
counterLevel	Campaign Definition	customerLevel
		cardLevel
		campaignLevel
counterType	Campaign Execution	RFM_local
		RFM_other
		Central
		Discount
PRODUCTS USING THE CORE ASSET		
Products	Constraints (Internal Variabilities)	
Cash Back Campaigns	No counters for Cash Back Campaigns.	
RFM Campaigns	RFM Campaigns have separate card level counters for local and other bank credit cards.	
Installment Campaigns	No counters for Installment Campaigns.	
Irregular Campaigns	No counters for Irregular Campaigns.	
Central Campaigns	Central Campaigns use the same counter for both local and other banks. For local bank, the counter level can be either card or customer. For other banks, the counter level can only be card.	
Discount Campaigns	Discount Campaigns have campaign level counters for only local bank.	

Table B.12: Sub-service decomposition for Campaign Counter Update

SUB-SERVICE DECOMPOSITION	
Sub-Service Name	Sub-Service Definition
<i>updateRFMCounter_local</i>	<i>Update the card level RFM campaign counter for local bank.</i>
<i>updateRFMCounter_other</i>	<i>Update the card level RFM campaign counter for other banks.</i>
<i>updateCentralCounter_card</i>	<i>Update the card level Central campaign counter for both local and other banks.</i>
<i>updateCentralCounter_customer</i>	<i>Update the customer level Central campaign counter for local bank.</i>
<i>updateDiscountCounter</i>	<i>Update the campaign level Discount campaign counter for local bank.</i>

Table B.13: Sub-service dependencies for Campaign Counter Update

SUB-SERVICE DEPENDENCIES ON VARIANTS				
SUB-SERVICES	VARIATION POINT: <i>bankCode</i>			
	<i>localBank</i>	<i>otherBank</i>		
<i>updateRFMCounter_local</i>	✓	✗		
<i>updateRFMCounter_other</i>	✗	✓		
<i>updateCentralCounter_card</i>	✓	✓		
<i>updateCentralCounter_customer</i>	✓	✗		
<i>updateDiscountCounter</i>	✓	✗		
SUB-SERVICES	VARIATION POINT: <i>counterLevel</i>			
	<i>customer Level</i>	<i>card Level</i>	<i>campaign Level</i>	
<i>updateRFMCounter_local</i>	✗	✓	✗	
<i>updateRFMCounter_other</i>	✗	✓	✗	
<i>updateCentralCounter_card</i>	✗	✓	✗	
<i>updateCentralCounter_customer</i>	✓	✗	✗	
<i>updateDiscountCounter</i>	✗	✗	✓	
SUB-SERVICES	VARIATION POINT: <i>counterType</i>			
	<i>RFM_local</i>	<i>RFM_other</i>	<i>Central</i>	<i>Discount</i>
<i>updateRFMCounter_local</i>	✓	✗	✗	✗
<i>updateRFMCounter_other</i>	✗	✓	✗	✗
<i>updateCentralCounter_card</i>	✗	✗	✓	✗
<i>updateCentralCounter_customer</i>	✗	✗	✓	✗
<i>updateDiscountCounter</i>	✗	✗	✗	✓

Table B.14: Product bindings for Campaign Counter Update

PRODUCT BINDINGS ON VARIANTS				
PRODUCTS	VARIATION POINT: <i>bankCode</i>			
	<i>localBank</i>	<i>otherBank</i>		
<i>Cash Back Campaigns</i>	X	X		
<i>RFM Campaigns</i>	✓	✓		
<i>Installment Campaigns</i>	X	X		
<i>Irregular Campaigns</i>	X	X		
<i>Central Campaigns</i>	✓	✓		
<i>Discount Campaigns</i>	✓	X		
PRODUCTS	VARIATION POINT: <i>counterLevel</i>			
	<i>customer Level</i>	<i>card Level</i>	<i>campaign Level</i>	
<i>Cash Back Campaigns</i>	X	X	X	
<i>RFM Campaigns</i>	X	✓	X	
<i>Installment Campaigns</i>	X	X	X	
<i>Irregular Campaigns</i>	X	X	X	
<i>Central Campaigns</i>	✓	✓	X	
<i>Discount Campaigns</i>	X	X	✓	
PRODUCTS	VARIATION POINT: <i>counterType</i>			
	<i>RFM_local</i>	<i>RFM_other</i>	<i>Central</i>	<i>Discount</i>
<i>Cash Back Campaigns</i>	X	X	X	X
<i>RFM Campaigns</i>	✓	✓	X	X
<i>Installment Campaigns</i>	X	X	X	X
<i>Irregular Campaigns</i>	X	X	X	X
<i>Central Campaigns</i>	X	X	✓	X
<i>Discount Campaigns</i>	X	X	X	✓

Table B.15: Traceability matrix for Campaign Counter Update

VARIANT BASED PRODUCT/SUB-SERVICE TRACEABILITY								
VARIANTS	SUB-SERVICES		PRODUCTS					
			Cash Back Campaigns	RFM Campaigns	Installment Campaigns	Irregular Campaigns	Central Campaigns	Discount Campaigns
VARIATION POINT: bankCode	localBank	updateRFMCounter_local	X	√	X	X	X	X
		updateRFMCounter_other	X	X	X	X	X	X
		updateCentralCounter_card	X	X	X	X	√	X
		updateCentralCounter_customer	X	X	X	X	√	X
		updateDiscountCounter	X	X	X	X	X	√
	otherBank	updateRFMCounter_local	X	X	X	X	X	X
		updateRFMCounter_other	X	√	X	X	X	X
		updateCentralCounter_card	X	X	X	X	√	X
		updateCentralCounter_customer	X	X	X	X	X	X
		updateDiscountCounter	X	X	X	X	X	X
VARIATION POINT: counterLevel	customerLevel	updateRFMCounter_local	X	X	X	X	X	X
		updateRFMCounter_other	X	X	X	X	X	X
		updateCentralCounter_card	X	X	X	X	X	X
		updateCentralCounter_customer	X	X	X	X	√	X
		updateDiscountCounter	X	X	X	X	X	X
	cardLevel	updateRFMCounter_local	X	√	X	X	X	X
		updateRFMCounter_other	X	√	X	X	X	X
		updateCentralCounter_card	X	X	X	X	√	X
		updateCentralCounter_customer	X	X	X	X	X	X
		updateDiscountCounter	X	X	X	X	X	X
	campaignLevel	updateRFMCounter_local	X	X	X	X	X	X
		updateRFMCounter_other	X	X	X	X	X	X
		updateCentralCounter_card	X	X	X	X	X	X
		updateCentralCounter_customer	X	X	X	X	X	X
		updateDiscountCounter	X	X	X	X	X	√
VARIATION POINT: counterType	RFM_local	updateRFMCounter_local	X	√	X	X	X	X
		updateRFMCounter_other	X	X	X	X	X	X
		updateCentralCounter_card	X	X	X	X	X	X
		updateCentralCounter_customer	X	X	X	X	X	X
		updateDiscountCounter	X	X	X	X	X	X
	RFM_other	updateRFMCounter_local	X	X	X	X	X	X
		updateRFMCounter_other	X	√	X	X	X	X
		updateCentralCounter_card	X	X	X	X	X	X
		updateCentralCounter_customer	X	X	X	X	X	X
		updateDiscountCounter	X	X	X	X	X	X
	Central	updateRFMCounter_local	X	X	X	X	X	X
		updateRFMCounter_other	X	X	X	X	X	X
		updateCentralCounter_card	X	X	X	X	√	X
		updateCentralCounter_customer	X	X	X	X	√	X
		updateDiscountCounter	X	X	X	X	X	X
	Discount	updateRFMCounter_local	X	X	X	X	X	X
		updateRFMCounter_other	X	X	X	X	X	X
		updateCentralCounter_card	X	X	X	X	X	X
		updateCentralCounter_customer	X	X	X	X	X	X
		updateDiscountCounter	X	X	X	X	X	√

B.4. Ledger Record Creation

Table B.16: Domain service description for Ledger Record Creation

CORE ASSET NAME	CORE ASSET TYPE	
ledgerRecordCreation	Domain Service	
CORE ASSET DESCRIPTION		
Creates records for general ledger in case of monetary outcomes while running campaigns.		
REQUEST PARAMATERS		
bankCode	Bank code of the credit card (localBank or otherBank)	
poolType	Type of the pool to be updated	
transactionInfo	Transaction information like date, channel, function, provision ID, etc. required for general ledger	
campaignID	Unique identification number of the campaign	
campaignType	1:Cash Back 2:RFM 3:Installment 4:Irregular 5:Central 6:Discount	
ledgerAmount	Reward amount in terms of TL for cash back or miles	
debitAccountID	Account ID in the general ledger for debit amount	
creditAccountID	Account ID in the general ledger for credit amount	
RESPONSE PARAMATERS		
responseCode	1:Ledger record creation successful, 0:Ledger record creation failure	
reasonCode	Reason code if ledger record creation failed	
VARIATION POINTS AND VARIANTS (External Variabilities)		
Variation Point	Source	Variants
bankCode	Campaign Execution	localBank
		otherBank
poolType	Campaign Definition	CCB
		PCB
		XCB
		Miles
		Lottery
PRODUCTS USING THE CORE ASSET		
Products	Constraints (Internal Variabilities)	
Cash Back Campaigns	Cash Back Campaigns create records for general ledger for the amount of cash back reward in terms of TL just for local bank. No ledger records are created for other banks.	
RFM Campaigns	RFM Campaigns create records for general ledger for the amount of cash back reward in terms of TL just for local bank. No ledger records are created for other banks.	
Installment Campaigns	No ledger records are created for Installment Campaigns.	
Irregular Campaigns	Irregular Campaigns create records for general ledger for the amount of cash back/miles reward in terms of TL just for local bank. No ledger records are created for other banks.	
Central Campaigns	Central Campaigns create records for general ledger for the amount of cash back/miles reward in terms of TL just for local bank. No ledger records are created for other banks.	
Discount Campaigns	Discount Campaigns create records for general ledger for the amount of discount reward in terms of TL just for local bank. No ledger records are created for other banks.	

Table B.17: Sub-service decomposition for Ledger Record Creation

SUB-SERVICE DECOMPOSITION	
Sub-Service Name	Sub-Service Definition
<i>createCashBackLedgerRecord</i>	Create ledger records for monetary outcome of cash back rewards.
<i>createMilesLedgerRecord</i>	Create ledger records for monetary outcome of miles rewards.
<i>createDiscountLedgerRecord</i>	Create ledger records for monetary outcome of discount rewards.

Table B.18: Sub-service dependencies for Ledger Record Creation

SUB-SERVICE DEPENDENCIES ON VARIANTS					
SUB-SERVICES	VARIATION POINT: <i>bankCode</i>				
	<i>localBank</i>	<i>otherBank</i>			
<i>createCashBackLedgerRecord</i>	✓	✗			
<i>createMilesLedgerRecord</i>	✓	✗			
<i>createDiscountLedgerRecord</i>	✓	✗			
SUB-SERVICES	VARIATION POINT: <i>poolType</i>				
	<i>CCB</i>	<i>PCB</i>	<i>XCB</i>	<i>Miles</i>	<i>Lottery</i>
<i>createCashBackLedgerRecord</i>	✓	✓	✓	✗	✗
<i>createMilesLedgerRecord</i>	✗	✗	✗	✓	✗
<i>createDiscountLedgerRecord</i>	✗	✗	✗	✗	✗

Table B.19: Product bindings for Ledger Record Creation

PRODUCT BINDINGS ON VARIANTS					
PRODUCTS	VARIATION POINT: <i>bankCode</i>				
	<i>localBank</i>	<i>otherBank</i>			
<i>Cash Back Campaigns</i>	✓	✗			
<i>RFM Campaigns</i>	✓	✗			
<i>Installment Campaigns</i>	✗	✗			
<i>Irregular Campaigns</i>	✓	✗			
<i>Central Campaigns</i>	✓	✗			
<i>Discount Campaigns</i>	✓	✗			
PRODUCTS	VARIATION POINT: <i>poolType</i>				
	<i>CCB</i>	<i>PCB</i>	<i>XCB</i>	<i>Miles</i>	<i>Lottery</i>
<i>Cash Back Campaigns</i>	✓	✓	✓	✗	✗
<i>RFM Campaigns</i>	✓	✓	✓	✗	✗
<i>Installment Campaigns</i>	✗	✗	✗	✗	✗
<i>Irregular Campaigns</i>	✓	✓	✓	✓	✗
<i>Central Campaigns</i>	✓	✓	✓	✓	✗
<i>Discount Campaigns</i>	✗	✗	✗	✗	✗

Table B.20: Traceability matrix for Ledger Record Creation

VARIANT BASED PRODUCT/SUB-SERVICE TRACEABILITY								
VARIANTS	SUB-SERVICES		PRODUCTS					
			Cash Back Campaigns	RFM Campaigns	Installment Campaigns	Irregular Campaigns	Central Campaigns	Discount Campaigns
VARIATION POINT: bankCode	localBank	<i>createCashBackLedgerRecord</i>	✓	✓	X	✓	✓	X
		<i>createMilesLedgerRecord</i>	X	X	X	✓	✓	X
		<i>createDiscountLedgerRecord</i>	X	X	X	X	X	✓
	otherBank	<i>createCashBackLedgerRecord</i>	X	X	X	X	X	X
		<i>createMilesLedgerRecord</i>	X	X	X	X	X	X
		<i>createDiscountLedgerRecord</i>	X	X	X	X	X	X
VARIATION POINT: poolType	CCB	<i>createCashBackLedgerRecord</i>	✓	✓	X	✓	✓	X
		<i>createMilesLedgerRecord</i>	N/A	N/A	N/A	N/A	N/A	N/A
		<i>createDiscountLedgerRecord</i>	N/A	N/A	N/A	N/A	N/A	N/A
	PCB	<i>createCashBackLedgerRecord</i>	✓	✓	X	✓	✓	X
		<i>createMilesLedgerRecord</i>	N/A	N/A	N/A	N/A	N/A	N/A
		<i>createDiscountLedgerRecord</i>	N/A	N/A	N/A	N/A	N/A	N/A
	XCB	<i>createCashBackLedgerRecord</i>	✓	✓	X	✓	✓	X
		<i>createMilesLedgerRecord</i>	N/A	N/A	N/A	N/A	N/A	N/A
		<i>createDiscountLedgerRecord</i>	N/A	N/A	N/A	N/A	N/A	N/A
	Miles	<i>createCashBackLedgerRecord</i>	N/A	N/A	N/A	N/A	N/A	N/A
		<i>createMilesLedgerRecord</i>	X	X	X	✓	✓	X
		<i>createDiscountLedgerRecord</i>	N/A	N/A	N/A	N/A	N/A	N/A
	Lottery	<i>createCashBackLedgerRecord</i>	N/A	N/A	N/A	N/A	N/A	N/A
		<i>createMilesLedgerRecord</i>	N/A	N/A	N/A	N/A	N/A	N/A
		<i>createDiscountLedgerRecord</i>	N/A	N/A	N/A	N/A	N/A	N/A

B.5. Transaction Log Creation

Table B.21: Domain service description for Transaction Log Creation

CORE ASSET NAME	CORE ASSET TYPE	
transactionLogCreation	Domain Service	
CORE ASSET DESCRIPTION		
Creates log records for all transactions which perform an update or insert operation on the database.		
REQUEST PARAMATERS		
bankCode	Bank code of the credit card (localBank or otherBank)	
transactionInfo	Transaction information like date, channel, function, provision ID, etc required for logging	
poolID	Unique identification number of the reward pool	
poolType	Type of the reward pool updated	
campaignID	Unique identification number of the campaign	
campaignType	1:Cash Back 2:RFM 3:Installment 4:Irregular 5:Central 6:Discount	
cardNo	Credit Card number performing the transaction	
customerID	Customer ID of the cardholder performing the transaction	
transactionAmount	Transaction amount in terms of TL	
rewardAmount	Reward amount in terms of cash back or miles	
RESPONSE PARAMATERS		
responseCode	1:Transaction log creation successful, 0:Transaction log creation failure	
reasonCode	Reason code if transaction log creation failed	
VARIATION POINTS AND VARIANTS (External Variabilities)		
Variation Point	Source	Variants
bankCode	Campaign Execution	localBank
		otherBank
PRODUCTS USING THE CORE ASSET		
Products	Constraints (Internal Variabilities)	
Cash Back Campaigns	Transaction logs are created for Cash Back Campaign rewards for local and other banks.	
RFM Campaigns	Transaction logs are created for RFM Campaign rewards for local and other banks.	
Installment Campaigns	Transaction logs are created for Installment Campaign rewards for local and other banks.	
Irregular Campaigns	Transaction logs are created for Irregular Campaign rewards for local bank only.	
Central Campaigns	Transaction logs are created for Central Campaign rewards for local and other banks.	
Discount Campaigns	Transaction logs are created for Discount Campaign rewards for local bank only.	

Table B.22: Sub-service decomposition for Transaction Log Creation

SUB-SERVICE DECOMPOSITION	
Sub-Service Name	Sub-Service Definition
<i>createLogRecord_local</i>	Create log records for transactions by credit cards of local bank.
<i>createLogRecord_other</i>	Create log records for transactions by credit cards of other banks.

Table B.23: Sub-service dependencies for Transaction Log Creation

SUB-SERVICE DEPENDENCIES ON VARIANTS		
SUB-SERVICES	VARIATION POINT: <i>bankCode</i>	
	<i>localBank</i>	<i>otherBank</i>
<i>createLogRecord_local</i>	✓	✗
<i>createLogRecord_other</i>	✗	✓

Table B.24: Product bindings for Transaction Log Creation

PRODUCT BINDINGS ON VARIANTS		
PRODUCTS	VARIATION POINT: <i>bankCode</i>	
	<i>localBank</i>	<i>otherBank</i>
<i>Cash Back Campaigns</i>	✓	✓
<i>RFM Campaigns</i>	✓	✓
<i>Installment Campaigns</i>	✓	✓
<i>Irregular Campaigns</i>	✓	✗
<i>Central Campaigns</i>	✓	✓
<i>Discount Campaigns</i>	✓	✗

Table B.25: Traceability matrix for Transaction Log Creation

VARIANT BASED PRODUCT/SUB-SERVICE TRACEABILITY								
VARIANTS		SUB-SERVICES	PRODUCTS					
			Cash Back Campaigns	RFM Campaigns	Installment Campaigns	Irregular Campaigns	Central Campaigns	Discount Campaigns
VARIATION POINT: <i>bankCode</i>	<i>localBank</i>	<i>createLogRecord_local</i>	✓	✓	✓	✓	✓	✓
		<i>createLogRecord_other</i>	✗	✗	✗	✗	✗	✗
	<i>otherBank</i>	<i>createLogRecord_local</i>	✗	✗	✗	✗	✗	✗
		<i>createLogRecord_other</i>	✓	✓	✓	✗	✓	✗