Evaluating The PLUSS Domain Modeling Approach by Modeling the Arcade Game Maker Product Line

Koteswar Rao Kollu
(ens03kku@cs.umu.se)

June 21st, 2005
Master's Thesis in Computing Science, 10 credits
Supervisor at CS-UmU: Magnus Eriksson
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

A Thesis submitted in partial fulfillment of the degree for Masters in Computing Science

ABSTRACT

Most published approaches for software product line engineering only address the software problems but not the systems problems. To tackle that problem the PLUSS Domain Modeling approach has been introduced at system level for requirements reuse within the systems engineering process. The PLUSS approach (Product Line Use case modeling for Systems and Software engineering) is a domain modeling method that utilizes Features, use cases and Use case realizations. An Arcade Game Maker Product Line example is used to evaluate the PLUSS approach. In this evolution the PLUSS notations for Feature Modeling and Use Case modeling are used to identify the similarities and variations between the three game products of an Arcade Game Maker Product Line. In this evaluation process some evolution criteria were defined and graded according to them. The results show that the PLUSS approach provides good overview of the domain with easily understandable documentation when compared with some standard notations of domain modeling. Hence the PLUSS approach is a good domain modeling approach and can be applied on any domain which is in the software product line strategy.

Acknowledgements

I would like to thank **Mr. Magnus Eriksson**, who supervised my thesis work with his advices and suggestions in the fulfillment of this thesis.

My special thanks to **Mr. Per Lindström**, for giving an opportunity to carry out my studies in Umeå University, Sweden.

Table of Contents

1. Introduction 1
- **1.1 Problem Statement 1**
- **1.2 Purpose 1**
- **1.3 Methods 2**
- **1.4 Arcade Game Maker Product Line Example 2**

2. Software Product Lines 4
- **2.1 Introduction to Software Product Lines 4**
- **2.2 Product Line Essential Activates 6**
- **2.3 The dimensions of Product Line: 7**
- **2.4 Developing a Product line architecture 8**
- **2.5 Product Line Practice areas 9**
- **2.6 Summery 10**

3. Feature Modelling 10
- **3.1 Introduction to Feature Modeling 10**
- **3.2 Feature Oriented Domain Analysis (FODA) 11**
- **3.3 Feature-Oriented Reuse Method (FORM) 11**
- **3.4 Feature based Reuse-Driven Software Engineering Business (FeatuRSEB) 12**
- **3.5 Generative programming (GP) 14**
- **3.6 Alexandria (Riebisch's) Notation 15**
- **3.7 Jan Bosch's notation 16**
- **3.8 Summery 17**

4. Unified Process 18
- **4.1 Rational Unified Process (RUP) 18**
- **4.2 Use Case Modeling 21**
- **4.3 Rational Unified Process for Systems Engineering (RUP SE) 22**

5 The PLUSS Approach 24
- **5.1 Introduction 24**
- **5.2 The PLUSS Feature Modeling 25**
- **5.3 The PLUSS Use Case Modeling 26**
- **5.4 The PLUSS approach to Modeling Variants in Use Case Models 27**
- **5.5 The PLUSS Notation for Describing Variants in Use case Specifications 27**

6. The PLUSS Evaluation Results 28

7. Conclusion 30

8. References 31

Appendix 1: Arcade Game Feature Diagram 33
Appendix 2: Arcade Game Feature Model Description 33
Appendix 3: Arcade Game Use-Case Specifications 33

List of Figures:

Fig 1: Documentation road map of Arcade Game Maker Product Line 3
Fig 2: Essential Product line Activities 6
Fig 3: Three decomposition dimensions of Software Product Lines 7
Fig 4: Feature diagram FODA: Car system 11
Fig 5: Feature categories 12
Fig.6: Feature diagram FeatuRSEB: phone service system 13
Fig 7: Example of a feature diagram in Generative Programming 14
Fig 8: Feature Diagram with multiplicities: Library system 16
Fig 9: Feature diagram of a mail client system...17
Fig 10: The Rational Unified process life cycle 18
Fig 11: The system use-case model for Arcade Game Maker Product Line........22
Fig 12: An example RUP SE Black box description 23
Fig 13: An example RUP SE White box step description 24
Fig 14: An example feature graph in the PLUSS notation 25
Fig 15: Feature constructs vs. Multiplicities in PLUSS 26
Fig 16: Blackbox flow of events used for describing use case scenarios 26
Fig 17: Whitebox flow of events for describing use case realizations 27
Fig 18: The PLUSS Meta-model 27
Fig 19: The PLUSS notation for describing variants in use case scenarios 28
Fig 20: The evaluation criteria..29
Fig 21: The PLUSS approach evaluation graph 29

1. Introduction

Software reuse is the most promising approach to increase the productivity and quality of software products [1]. In traditional software reuse, a library of reusable code components was developed. Studies have shown that instead of reusing an individual component, it is much more advantageous to reuse whole system design or subsystem [1]. This leads the basic idea of Product lines. A product line is defined as [3] “A set of systems that share a common and managed set of features satisfying the specific needs of a particular market segment”. The basic idea of product line approach is to use domain knowledge to separate the common parts of a family of products from the differences between the products. One notation used for modeling commonalities and variants with in system family is known as feature modeling [7]. Feature models are applied to describe variable and common properties of products in a product line with an overview over requirements and differences between features. Several feature models are used for development and application of software product lines. Eriksson et al, introduced a use case driven approach for product line development in [6, 7] known as the PLUSS approach. In this thesis, the PLUSS approach is applied on a product line example, known as the Arcade Game Maker Product Line [16]. The purpose of this was to evaluate how suitable the PLUSS approach is for that particular domain.

This thesis report is divided into seven sections. In section 1 the problem addressed, main goal of this thesis and introduction to the Arcade Game Maker Example are described. Section 2 describes the framework on Software Product Lines and section 3 describes the concepts and several notations of Feature Modeling. The Rational Unified Process for Systems Engineering and basics are discussed in section 4. The concepts and notations of PLUSS approach is described in section 5. The results of evolution are discussed in section 6 and finally section 7 gives the conclusion of this thesis. The appendix 1 is the derived feature model for online Arcade Game Maker Product Line example; appendix 2 is a description of that feature model and in appendix 3 shows the scenarios for the Use cases and Change cases of Arcade Game Maker Product Line example.

1.1 Problem Statement

The main goal of this thesis is to evaluate the domain modeling approach called ‘the PLUSS approach’ by applying it on the online Arcade Game Maker Product Line example. The basic idea of Software Product Lines, Feature Modeling and Rational Unified Processing for Systems Engineering (RUP-SE) were investigated to understand the concepts and notations of the PLUSS approach. The architecture of the Arcade Game Maker Product Line example and its related documents must be reviewed to find out the relations and dependencies among products and its features to apply the PLUSS notations. Finally with this experience the PLUSS approach is evaluated with evaluation criteria and grades.

1.2 Purpose

The purpose of my thesis is to study the standard product line architecture for Arcade Game Maker Product Line example and applying the PLUSS approach on it to derive the standard feature model based on PLUSS notations by combining features and use

cases of the Arcade Game Maker Product Line example and also to learn the basic concepts of PLUSS approach and its application in a particular domain.

1.3 Methods

The following step wise methodology was used for the PLUSS evaluation process.

1. Initially I started with the literature survey of software product lines; feature modeling, Rational Unified Process for Systems Engineering (RUP-SE) as those are related to the PLUSS approach.
2. The Arcade Game Maker Product Line example and its documents were then analyzed to find out the product line structure, number of products and each product feature.
3. In this step the PLUSS approach and its notations were studied, with the basic knowledge of software product lines, feature modeling and RUP-SE, made the PLUSS notation easy to understand.
4. Later the PLUSS approach and its notations are applied on the Arcade Game Maker Product Line example. In this process the existing feature model of Arcade Game Maker Product line is re-structured according to the PLUSS feature model notation. The relationships and dependencies are defined between features. The use case scenarios and change case scenarios are used for deriving the black box flow of events as shown in appendix 3. The use cases were then structured around the feature model in accordance with the PLUSS approach and features of the feature model are described for the traceability purpose.
5. Finally, with the above experience the evaluation of the PLUSS approach has done with the following evaluation criteria, with which several important aspects of the PLUSS approach is examined and graded.
    - **Ease of Learning**: The process of learning and understanding the concepts and notations of the PLUSS approach.
    - **Richness of Notation**: How well the notations of the PLUSS approach describe the solution.
    - **Tool Support**: How well the various MS-Office tools supported to draw the feature diagrams and use case realizations of the PLUSS approach.
    - **Needed Modeling Effort:** The domain modeling knowledge to use the PLUSS approach.
    - **Usefulness for Understanding the Domain:** How easily a domain can be understood by the analysts with the help of the PLUSS approach.

1.4 Arcade Game Maker Product Line Example

As discussed in [16], the Arcade Game Maker product line is an example for demonstrating and learning the concepts of software product lines. This product line consist three simple arcade games Brickles, Pong and Bowling. These are based on single system architecture. This example is used as a case study for implementing the software product line approach. This case study is an estimation to the company that implement the product line approach and provides a number of assets that are used in the development of product line architecture.

Fig 1 provides an overview of the available documentation in the Arcade Game Maker product line example. Each asset is described individually as follows:

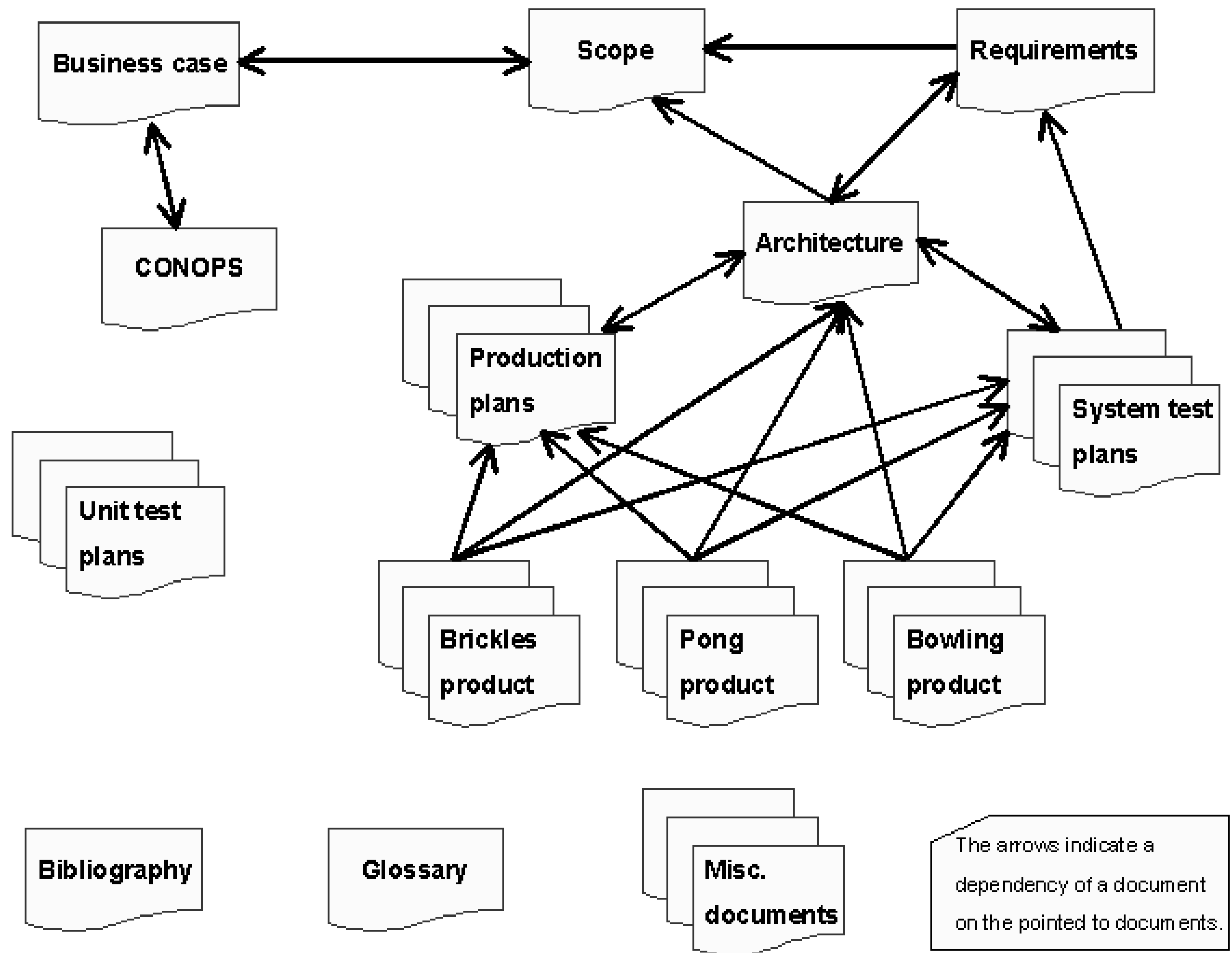


Fig 1: Documentation road map of Arcade Game Maker Product Line [16]

Business Case
This document provides an overview of adapting software product line strategy for the products of a company. It also provides information on how a company can achieve all the benefits of a product line approach and the impact of changes that enable the hidden costs and benefits to the company to adapt product line approach.

Scope
This document describes the design and implementation decisions that are made within the boundaries of product line. This document is useful for architects to find reusable components of products in the product line and managers to manage the product planning.

Requirements
One of the most important documents in any software product development, which provides information to the analysts, stakeholders, managers and designers in all the phases of development process. This document focuses on the functional and non-functional requirements and their implementation process. In a product line approach the functional requirements are derived by using the variability and commonality

among products and its features. The feature model and use case scenarios are developed using this document to address the requirements.

Architecture
This document is a route map to the implementation of products, which describes modules and interface that is to be implemented for the total system development. This document provides the risks, tradeoffs, sensitivities that are associated with the development process.

Production Plans
This document provides the production strategy and core assets associated with production. It also describes the product qualities.

System Test Plans
Every software product is tested to find-out faults of the system and to verify that requirements are meet. This document provides the testing items and testing strategy. Production teams use this document to test the products iteratively.

Concept of Operations (CONOPS)
This document used by product line organizations to make decisions and to manage the production work, it describes the organizational and technical considerations.

Brickles Product
This document contain all the information about brickles product like user's manual, system test plan, production plan, program executable and structure of product line code for brickles product. The user's manual contains the information about game operation, rules and expected output. The product line code is the combination individual packages of the product.

Pong Product
This document contains the information about Pong product, its user's manual, system test plan, production plan, product line code and executable version of Pong product. The user's manual contains information about game, its operation, rules and expected output.

Bowling Product
This document contains the information like user's manual, product plan, system test plan, product line code and executable version of bowling product. The user's manual contains information about game, its operation, rules and expected output.

2. Software Product Lines

2.1 Introduction to Software Product Lines

According to [1], several approaches have been proposed to overcome the Software crisis. During 1960s software systems was developed by integrating components and in 1970s several module based approaches were proposed. With the introduction of Object Oriented Programming in 1980s classes are used as units of reuse. All these approaches only provide reuse at the individual level and in small-scale. As a result of

this, Software Product Lines were introduced in early 1990's by combining the software architecture and component based software development.

The product lines strategy is widely used in the manufacturing industry since a long time to raise the production capability and reduce time to market by designing a set of products to have many parts in common. The Software Product Line development strategy enables an organization to make optimal use of resources by setting up a strategic platform for software development [17]. Under product line strategy a wide variety of products are developed and maintained very efficiently. The Product line practice in software development will increase the productivity and quality of a software product with high levels of reuse. The planed selection of similar products will boost the economic stability of a software development company. A software product line is defined as:

"A software product line is a set of software intensive systems that share a common, managed set of features satisfying the particular market segment or mission that are developed from a common set of core assets in a prescribed way" [17]

In traditional software development each system is built individually, but where as in software product lines a family of software systems are considered. Systems developed with product line approach are capable of adapting to changes in requirements during each iteration provided those changes must be in the same domain of a product line. As discussed in [3], many organizations are realising that a product line of software systems built from common set of assets could result in increased market share, greater customer satisfaction, higher system reliability, and lower the staffing requirements. This commonality enables the multiple kinds of benefits for an organization and these are categorised as strategic business benefits, engineering benefits and personal benefits. The strategic business benefits are reuse of a multiple shared assets, including architecture, reusable components, schedules, budgets, test cases, performance models, documentation, marketing plans and literature. Apart from these, the other possible engineering benefits in production area are, according to [11]:

- The average time to create and deploy a new product will reduce
- The average number of defects per product will be reduced
- The average engineering cost per product will go down because of the reduction of engineering effort to deploy and maintenance of a product
- The total number of products will increase with effective management

The individual (personal) benefits according to [3] are:

- The CEO, benefits economically by capturing the new markets with large-amount of productivity and better time to market.
- The COO, can easily manage the huge work force in an efficient way, can easily allocate the employees among the various locations because of commonality of the applications.
- The Technical Manager, benefits from the forecasting of schedules, roles and responsibilities of the workforce.

- The Software Product Developer, benefits the greater job satisfaction and ease of schedules and more time to concentrate on current products and new technology.
- The Architect or Core Asset Developer, can expertise the skills and greater challenge in the job for designing the software that will be used for many products.
- The Marketer leads the organization into greater profits by selling high quality and more predictable delivery products.
- The Customer, benefits from the high quality of products with well tested training materials and documentation. Predictable delivery date and lower product cost with less maintenance costs.
- The End user, benefits better and easy learning materials, documentation

2.2 Product Line Essential Activates

Before going to essential product line activities the core assets are defined as [17]:
"An artifact or resource that is used in the production of more than one product in a software product line". A core asset may be architecture, a software component, a process model, a plan, a document, or any other useful result of building a system.
The domain of product line development involves core asset development and product development using core assets. There is strong feed back connection between core assets and the products. Core assets are refreshed as new products are refreshed. Management plays a vital role to view new products is in the context of the available core assets. The three essential activates (see fig: 2) are described individually as discussed in [17]

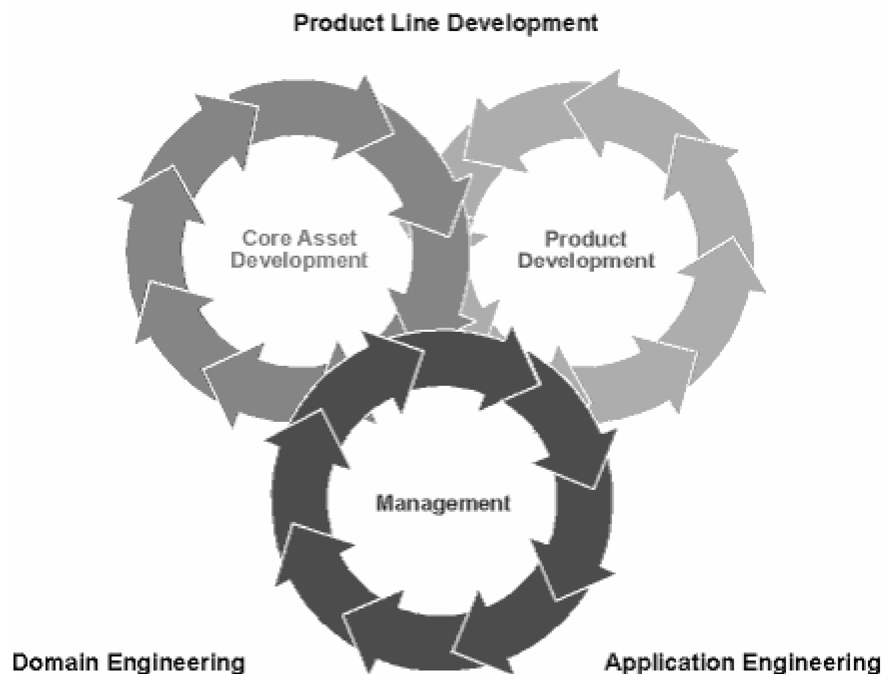


Fig 2: Essential Product line Activities [17]

1. Core Asset Development
The main goal of core asset development activity is to provide a base for production or products. The core asset developers provide a range of assets like architectures, plans, specifications and implementations to the product developers. Plans include test plans and production plans and templates for production development

2. Product Development
The product development activity depends on the requirements of a particular product and the outcomes of core asset development. Requirements are expressed as delta from set of product line requirements. Product line scope indicates when products can or cannot be included in product line. Product plan gives details about how assets are used to build products.

3. Management
The Management activities are divided into technical and organizational. Both levels must be strongly committed to the software product lines. Technical management manages the core asset development and the product development activities by checking the groups that build core assets and products are engaged in required activities. Organizational management can be defined as the authority that is responsible for the ultimate success or failure of the product line effort.

2.3 The dimensions of Product Line:

The concepts of software product lines are decomposed into three dimensions as shown in fig 3:

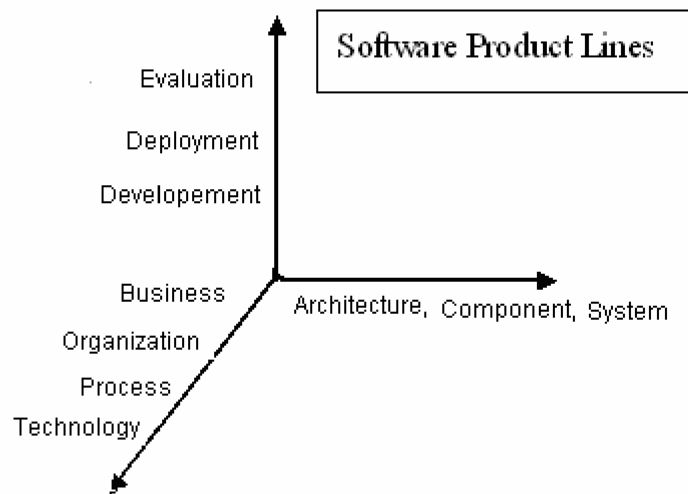


Fig 3: Three decomposition dimensions of Software Product Lines [1]

The first dimension represents the primary assets that are part of the reuse-based development i.e. Architecture, Component and System. The different views of an organization describe the second dimension as Business, Organization, Process and Technology. The third decomposition is based on the lifecycle of each of the assets in the organization are Development, Deployment, and Evolution.

2.4 Developing a Product line architecture

According to [1], the design phase of product-line software architecture contains six basic steps, i.e.

Business case analysis

The primary aim of this step is to identify the benefits and hidden costs of adopting product-line approach compared with presently used approach. The business case analysis is done in four steps, starting with analysis of current situation in an organization. Secondly, a prediction of future costs and benefits of current approach are assumed. Thirdly the investment required to convert product line to be analyzed. Finally the benefits and future costs of product line strategy are analyzed. The costs and benefits can be pure financial figures, man hours, time or logical combination of all.

Scoping

This activity is divided into several steps i.e. product selection, feature selection, feature-graph specification, product-line scoping and the product-specific requirements. All the above steps conclude with what are products and product features that are to be included and excluded in product line from the starting.

Product and feature planning

The product and feature planning is an extension of scoping activity with requirements related to the ease of incorporating predictable features and products. This plan can be used by software architects to prepare the product line architecture for future inclusion of other features.

Design of Product-line architecture

This is the main step in the process of developing software product-line architecture. There are several approaches to design product line architecture. In product line architectures the products are expressed using variability of the components. Every architectural design consists three phases i.e. functionality based design, architecture assessment and architecture transformation. Product specific features are to be considered while designing product line architecture to maintain the quality of product.

Component requirement specification

The goal of this activity is to specify the requirements specification for each component. The requirements specification contains several aspects that are to be defined with each component are interfaces, functionality, quality attributes and variability.

Validation

It is important to verify the product-line architecture with the requirements of the stakeholder before developing components. This can be done by meeting with stakeholder or by architecture assessment teams to identify weakness in software architecture for an individual product or a product-line.

2.5 Product Line Practice areas

The Software Engineering Institute has identified 24 practice areas as skills that are need for an organization to adopt the product line strategy. 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” [17].
Practice areas provide starting points from which organizations can make progress in adopting a product line approach for developing software products. These practise areas loosely divided into three categories, as stated below.

1. Software Engineering practice areas
2. Technical Management practice areas
3. Organizational Management areas

Software Engineering practice areas
These practise areas are necessary for an application of suitable technology to the creation and evaluation of the core assets and products.

- Architecture definition
- Architecture Evaluation
- Component Development
- COTS Utilization
- Mining Existing Assets
- Software Systems Integration
- Testing and
- Understanding Relevant Domains.

Technical Management practice areas
Technical management practice areas are carried out in the technical activities represented by the core assets and product development parts.

- 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
These practice areas are necessary for the orchestration of the entire product line effort and these practice areas are concerned only with the management.

- 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

2.6 Summery

As software industry is growing rapidly the generation of similar application systems in a domain are made easy and cost effective with the help of software product line architecture. Successful product line architecture involves the systematic management of planned variations across the product line. The commonality permits the reuse of assets in key areas like architecture, schedules and budgets, test case, marketing plans, literature, training and documentation. The software product line strategy is composite of software engineering aspects of product production with organizational and technical management and many number of practice areas are derived to ease the adaptation of product line process for many organizations.

3. Feature Modelling

3.1 Introduction to Feature Modeling

The introduction of the software product-line approach enabled development of huge and multiple software applications with high levels of software reuse. Domain analysis techniques were proposed to reduce the risk of developing inappropriate software because of unknown future requirements. Domain analysis reduces the risks by analyzing concepts, properties and solutions of a domain. Feature models are used in domain analysis to provide an overview over the requirements. In domain modeling the common and variable requirements for software systems are described as instances of a product line. A Feature model contains a feature diagram and some additional information, such as relationships and dependency among product features. A feature model also shows the functionality that can be selected when building new systems in the domain. Feature models provide an abstract and syntax for expressing commonality and variability in the domain. The feature model resides between requirements model and design model [19].

Feature Diagrams

A feature diagram is a hierarchical decomposition of features with the indication of feature types. A feature diagram constitutes a tree composed of root, nodes and directed edges. The root represents a concept and the rest of the nodes represent features. Edges connect concept with its features and a feature with its sub-features. Feature diagrams describe relations between various requirements and distinguish between common and variable characteristics of a concept. The concept refers to a property of a product or domain. A feature can be included in a concept instance only if a parent has been included. Feature diagrams are important product of domain analysis in product line strategy.

A number of notations have been proposed in the literature, examples are FODA [12], FORM [13], FeatuRSEB [8], Generative Programming [4], ALEXANDRIA [20] and Jan Bosch's [2] notation.

3.2 Feature Oriented Domain Analysis (FODA)

The first feature notation FODA, method, was introduced in 1990. According to [12], "A feature is a prominent or distinctive user visible aspect, quality or characteristic of a software system or systems"

Feature Oriented Domain Analysis (FODA) is a domain analysis and engineering technique which focuses on developing reusable core assets for multiple products in the domain. Domain analysis is "the process of identifying, collecting, organizing and representing relevant information in a domain based on the study of existing systems and their development history" [12].

FODA feature models describe mandatory, optional and alternative properties of concepts within domain. A filled circle at the top of the feature identifies a mandatory feature. A mandatory feature must be selected in all the systems of a domain. An empty circle at the top of the feature identifies as optional feature. Optional features are only present in the application if the customer has chosen them. An arc spanning two or more edges of the feature nodes depicts as set of alternative features. The term alternative feature indicates that a system can possess only one sub-feature at a time for main feature. As shown in fig 4 the example of car feature model, transmission and horsepower are mandatory and must be selected, where as air conditioning is an optional feature. The transmission has two alternatives in which one must be selected either manual or automatic for an instance of transmission.

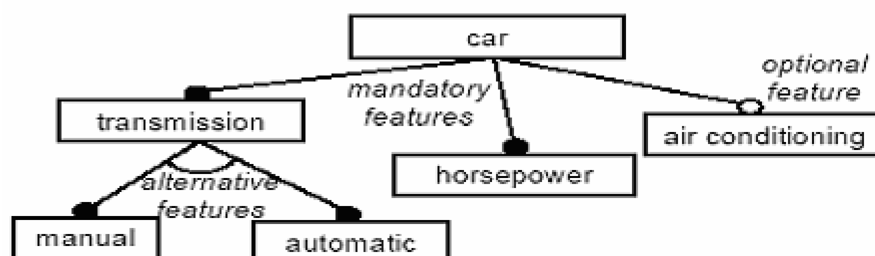


Fig 4: Feature diagram FODA: Car system [12]

3.3 Feature-Oriented Reuse Method (FORM)

The FORM [13] is the prolonged study of FODA method. FORM starts with an analysis of commonality among the applications in a particular domain. In the FORM product features are identified and classified in terms of Capabilities, Domain technologies, Implementation techniques and Operating environments as shown in the fig 5. Capabilities are end user-visible characteristics and can be identified as System services, Operations and Non-functional characteristics that are specified by the customer. Domain technologies are domain specific methods and problem solutions that are used by domain experts to represent the way of implementing services. Operating environments represent an environment in which the applications are operated like hardware environment and software environment. All the components of the system with their interfaces and protocols are part of this category.

Implementation techniques are general problem solutions in which domain functions, services and operations are implemented.

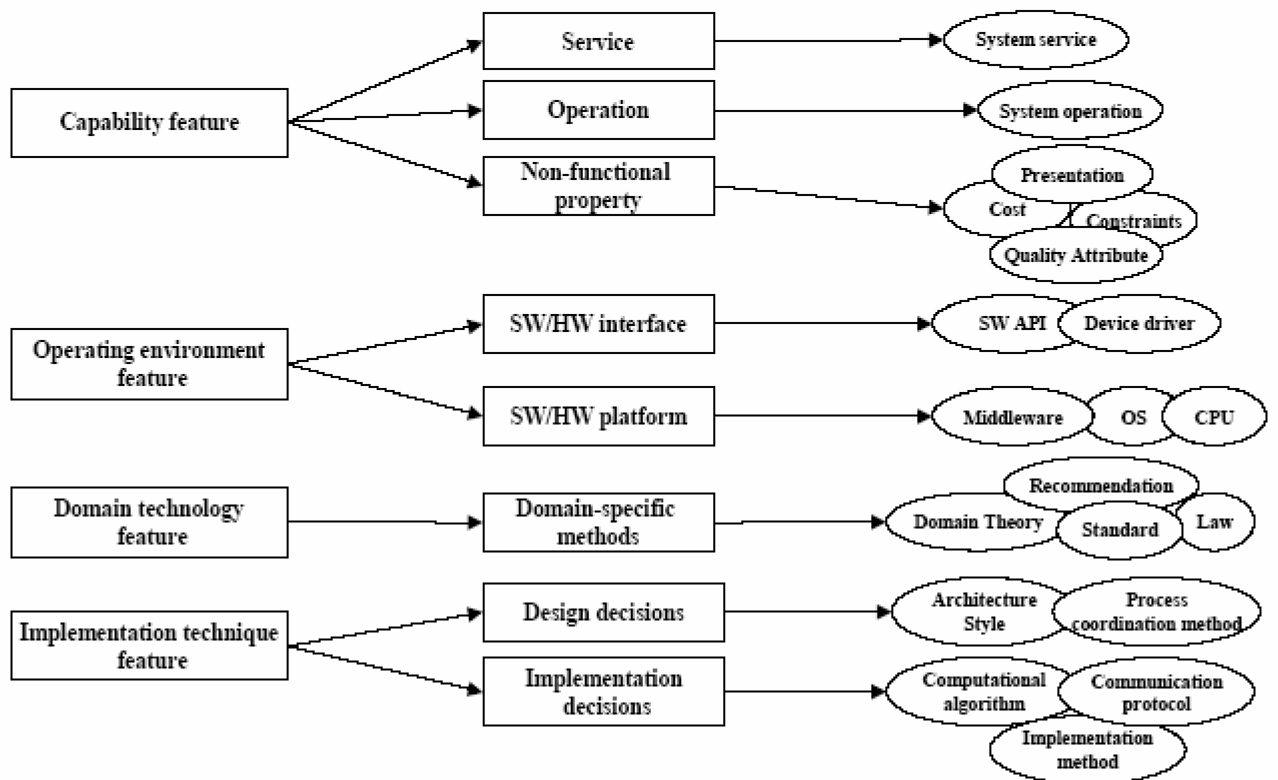


Fig 5: Feature categories [14]

In FORM the common features in different products are mandatory features, variable between different products are named as optional or alternative. Optional features represent selectable features for products in the domain. In alternative features only one feature is selected for an instance. The generalization /specialization relationship can be used when features are generalised with sub-features. When one feature is directly depend on other future the “implemented-by” relationship is used to represent those two features.

3.4 Feature based Reuse-Driven Software Engineering Business (FeatuRSEB)

FeatuRSEB [8] is the integration of FODA [12] and Reuse-Driven Software Engineering Business (RESB). The RSEB is a use-case driven systematic reuse process. In RSEB, a use case and analysis object model is used to describe system architecture and context. In RESB the variability is captured using explicit variation points and variants. The RSEB method uses features informally; features are related to use cases or parts of use cases. Unlike FODA, RSEB provides no explicit feature models that construct and transform such feature models. As discussed in [8], the feature model construction process can be outlines as:

1. Merge individual use case models into a domain use case model, capture and express the differences by using variation points.
2. Create an initial feature model with functional features derived from the domain use case model.

3. Create the RSEB analysis object model, augmenting the feature model with architectural features.
4. Create the RSEB design model augmenting the feature model with implementation features.

The example shown in fig 6, of the feature model for Rapid Telephone Service Creation, illustrates the major relationships in the FeatuRSEB feature model.

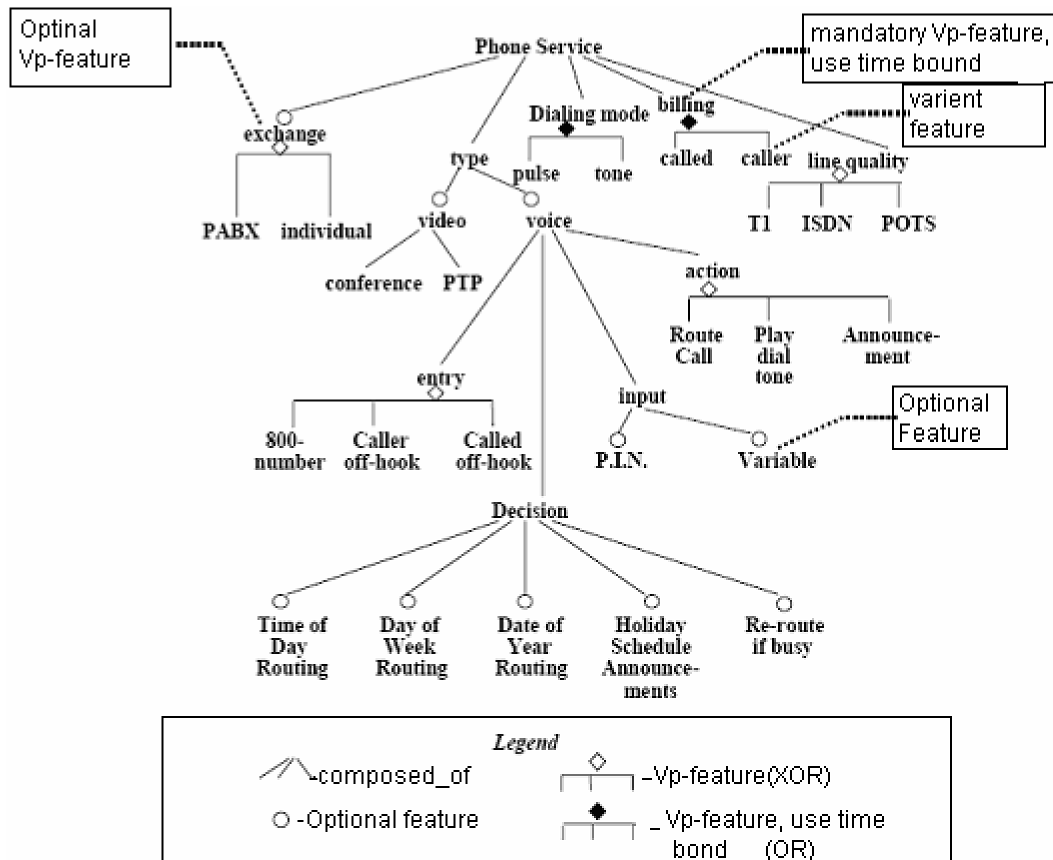


Fig.6: Feature diagram FeatuRSEB: phone service system [8]

The above shown features can be specified by using the following notations

1. The composed_of relationship: A feature can be modelled as composed of several sub features. For example in fig 6, the feature “phone service” is composed of “exchange”, “type”, “dialling mode”, “billing”, and “line quality”. This relationship is represented by a line from super-features to each of its sub-features.

2. The existence attribute: Determines whether a feature is mandatory or optional. An optional feature represented with circle about the feature name.

3. The alternative or XOR relationship: A feature can act as variation point (vp-feature) in the model and other features can be defined as variants. In the example the feature “exchange” is a vp-feature with “PBX” and “individual” as variants. Only one of them is selected and these features bind at use time. A variation point is represented with a diamond under its name. A line is drawn to each available variant from the diamond.

4. The OR-relationship: Defines a feature as a variation point and other features as its variants from which one or more joined. A variation point feature is represented by

a black diamond under its name. A line is drawn to each available variant from diamond.

3.5 Generative programming (GP)

Generative Programming is defined as “A software engineering paradigm based on modelling software system families such that, given a particular requirements specification, a highly customised and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge”[4].

As described in [4], the feature model in GP defines the scope and configurability aspect of a system family and provides the configuration knowledge needed to automate the production of family members. The GP-feature diagram slightly extends the FODA notation with or-features. A mandatory feature is included in the description of a concept instance if and only if its parent is included in the description of the instance. A mandatory feature is represented with a simple edge ending with a filled circle. An optional feature may or may not be included if the parent is included. An optional feature node is represented with a simple edge ending with an empty circle. An alternative feature is a feature from a set of features from which only one can be chosen if the parent of a set of alternative feature included. An or-feature is the nodes of a set of alternative features are pointed to by edges connected with a filled arc. A feature may have one or more sets of direct or- sub features. An or-feature can also be an optional.

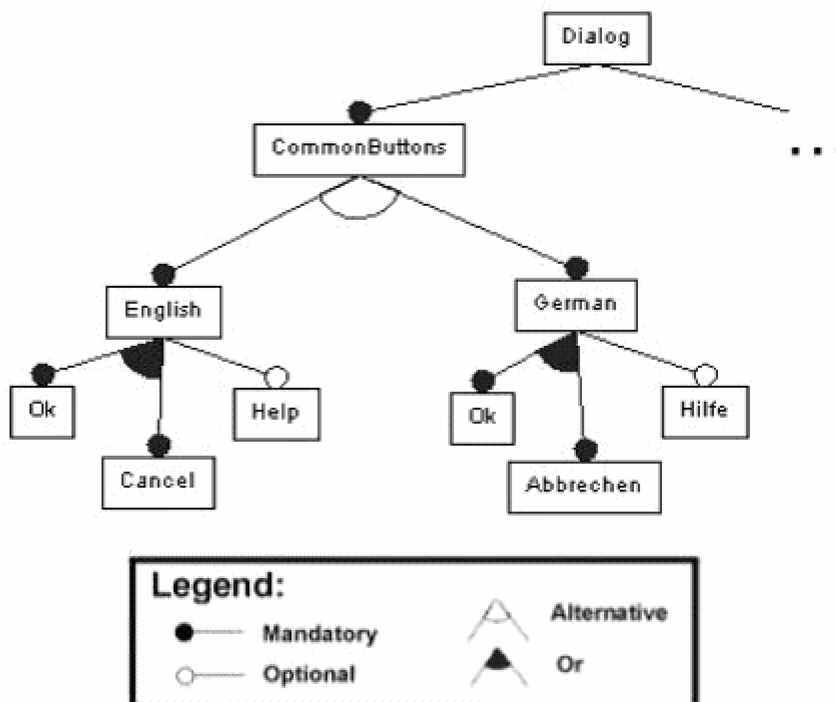


Fig 7: Example of a feature diagram in Generative Programming [22]

The example of feature diagram fig 7, describes a part of dialog window. The root represents the dialog concept and the other features as described as:

Mandatory features**:** Every dialog has the common buttons
Alternative features**:** A dialog window may support either English or German languages
Or-features**:** A dialog window may have an Ok button, Cancel button or both.

3.6 Alexandria (Riebisch's) Notation

This notation has been introduced to prevent the ambiguity and to refine the relation between features of Generative Programming. Alexandria is methodology for developing and evolving software product lines and developed by Reibisch et al [20]. In this notation feature diagrams are extended with UML multiplicities and are defined between neighbouring features of a feature diagram.

Multiplicity	Definition
0..1	at most one feature has to be chosen from the set
1	exactly one feature has to be chosen from the set
0..*	an arbitrary number of features (or none at all) have to be chosen from the set
1..*	at least one feature has to be chosen from the set
n..*	n number of optional features can be selected

As discussed in [20], the fundamentals of this notation are:

- A feature is a node in a directed –acyclic graph.
- Relations between features are expressed by edges between features. A circle at the end of its corresponding edge determines the direction of a relation.
- If this circle is filled, then the relation between features is said to be mandatory, i.e. when the feature at the relation's origin is chosen, the feature at the relation's destination has to be chosen, too.
- If the circle is empty, the relation is non-mandatory i.e. the features at the relation's destination need not to be chosen, it is optional.
- Optional relations that originate from the same feature node can be combined into a set. Each relation can only be part of one set.
- A set has a multiplicity that donates the minimum and maximum number of features to be chosen from the set. The possible multiplicities are: 0..1, 1, 0..n, 1..n and n..*. Visually a set is shown by an arc that connects all the edges that are part of the set. The multiplicity is drawn in the center of the arc.
- Relations between features those are not located in the adjacent parts of the graph shown separately because of the clarity of the diagram. Such relations can be described in a textual form by using subset of UML Object Constraint Language (OCL) [20].

The following example shown in fig.8 is a feature diagram of a Library system in the Alexandria notation. Three features with sets were identified in the library system family.
First set is managing books with different items like Book, Journal and/or Audio book. At least one of these items must be managed by the system otherwise no customer can borrow books from the library. In the second set, if the customer wants to borrow a book, he has to identify himself to the librarian either by using a chip card or biometric way (fingerprint check). When developing the system one of the two alternatives has to be chosen. The third set is customer's data, every time a new

customer registers himself at the library and is required to authenticate himself to the librarian if he has lost his chip card or wants to prong his book by phone.

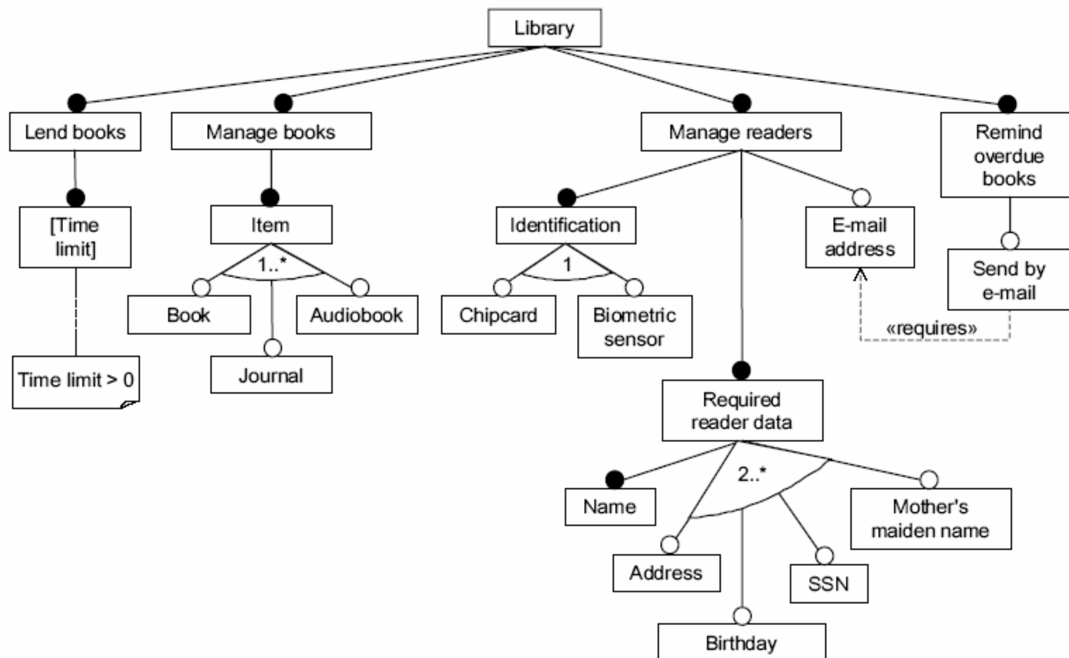


Fig 8: Feature Diagram with multiplicities: Library system [20]

3.7 Jan Bosch's notation

According to [2], a feature is defined as "a logical unit of behaviour that is specified by the set of functional and quality requirements".
This notation is slightly different from FeatuRSEB [8] notation, with the addition of binding times and external features. In this notation:

- Features are represented by rectangle (like in FORM)
- External features are represented by dotted rectangle.
- A composition construct is used to group related features.
- The alternative or XOR-relationship is represented by a non-filled triangle
- The OR-relationship is represented by black-filled triangle

The example feature diagram of a mail client system is shown in fig 9 and the features are categorised as following [2]:

External features
These features are not direct part of the system but offered by the target platform of the system. These features are essential because system uses and depends on them.
For example in the mail client system, "TCP Connection" is essential to connect to other computer but not a part of client requirement.

Mandatory features
These are the features that identify a product, for example in an email client system; "Type in Message" is a known feature to type text and send as email.

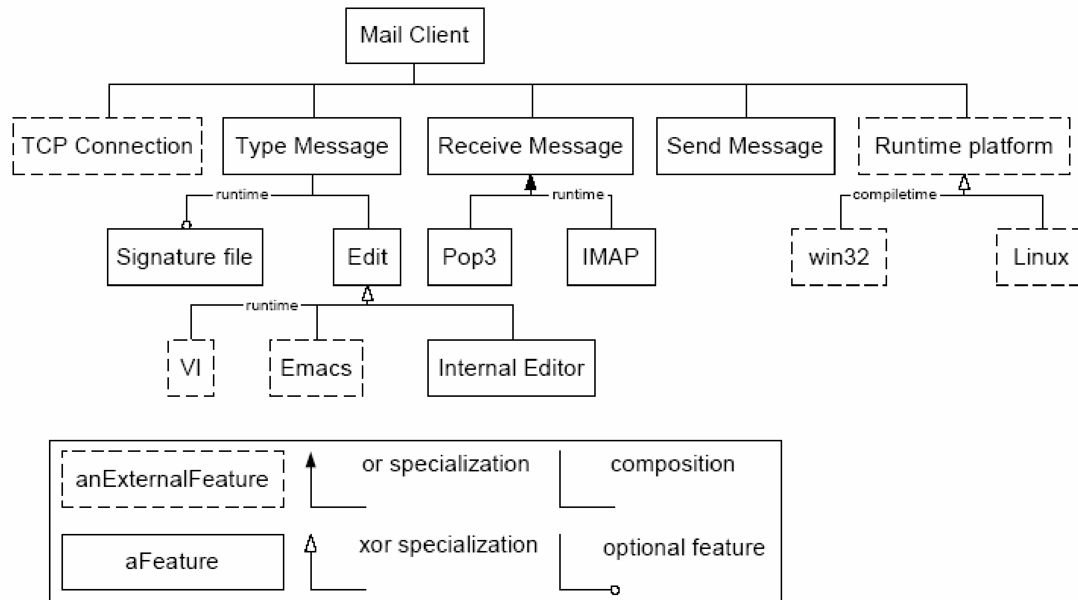


Fig 9: Feature diagram of a mail client system [2]

Optional features
These features are embedded with some core features and optional, for example in email client system "Signature file" are optional and can be used by any user who wishes to send signature with each email.

Variant features
A variant feature is an abstraction for a set of related features (optional or mandatory). For example in email client system, client may use and configure any kind of editor to type and edit text.

3.8 Summery

With the introduction of the first feature model notation by Kang et al. in FODA [12], feature models made it easy for stakeholders to understand the abstract view of the product family with feature descriptions. Since then feature models are used to define the products and product configurations with the addition of new products to the existing software product line architecture. A feature model provides an overview over requirements, dependencies and relations between features. The feature modelling is used to model the differences between commonality using variable properties of a product line. FODA [12] notation is clear and easy to understand but it does not have enough expressiveness to explicitly represent variation points [9]. The four layers of FORM [13] describe different views related to the product development, but it generates the complexity when large number of variants represented. FeatuRSEB [8] is the first idea to combine both features and use cases to explicit representation of variation points. In GP [4] the or-features are described as alternative-optional features, but the relations between features leads to ambiguity [20]. In Reiebish's notation [20] the feature diagram is more simplified but the variation points not explicitly defined even if cardinalities can be identified. Finally

J.Bosch's notation [2], replaces the non filled and black filled diamond of FeatuRSEB [8] with XOR and OR relationships.

4. Unified Process

4.1 Rational Unified Process (RUP)

According to [10], The Unified Process is a component based software development process. The Unified Process uses the Unified Modeling Language (UML) for preparing software system blue prints. The Rational Unified Process (RUP) is a commercial instance of the Unifies Process which is developed and maintained by the IBM-Rational Corporation.
According to [15] The RUP is a software engineering process that would improve the productivity of software development for larger software systems. The goal of RUP is to produce high quality software with specified end user requirements in predictable schedule and budget. The Rational Unified Process attempts to improve the team productivity by providing access to the knowledge base for every team member. The RUP supports various visual modeling, programming, and testing tools. The RUP activities create and maintain models. A model is a set of plans that describe the complete system from a particular perspective. In RUP, one development cycle is described in two dimensions as shown in the below Fig 10. The horizontal axis represents the time and dynamic aspects of the process and it is represented in phases, iterations, milestone and cycles. The vertical axis represents the static aspects of the process. In this view the process is described in activities, artifacts, workers and workflows. A phase is defined as a time span between two major milestones in the software development process. A milestone is "A point in time at which certain critical decisions will be made" [15].

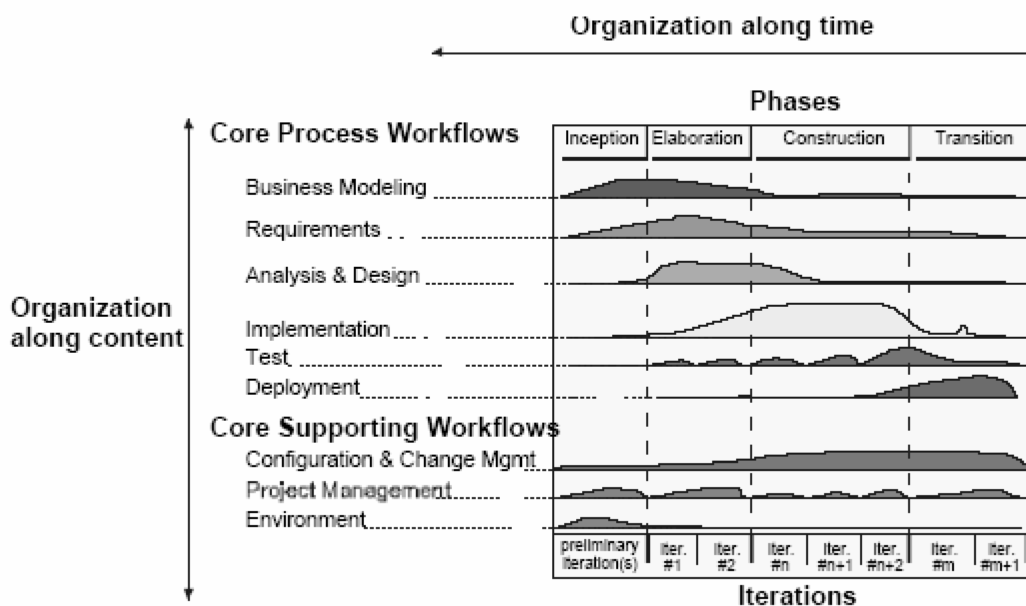


Fig 10: The Rational Unified process life cycle [15]

The RUP divides one development cycle in four consecutive phases. Each phase of the RUP can be further broken down into iterations. An Iteration is "The complete

development loop resulting in a release of an executable product, a subset of the final product under development, which grows incrementally from iteration to iteration to become the final system" [15].

Inception phase
In this phase the scope of the project is defined and business case is developed for the system. The scope of the project includes identifying all use cases and specifying most important ones with actors. The business case includes risk assessment, success criteria, resource estimation and duration of the phase plan.

Elaboration phase
This is the most critical phase among all phases. It includes analysis of domain problem, development of the project plan, specifying features, and an executable architecture is developed in one or more iterations depending on the risk, scope, size, and novelty of the project.

Construction phase
During this phase remaining components and application features developed and integrated into product. All features are tested thoroughly finally in this phase product is build.

Transition phase
The software product is transitioned to the user community. This phase focuses on the activities required to place the software into the hands of the users. The activities include beta releases, general availability releases, several iterations and enhancement releases.

The RUP is represented using four primary modeling elements: Workers, Activities, Artifacts and workflows explained each below.

Workers
A worker defines the behavior and responsibilities of an individual or a group of individuals working together as a team.

Activities
An activity is performance of workers when executing tasks. Every activity is assigned to a specific worker.

Artifacts
Artifacts are the outcomes of the activities and an artifact is a piece of information that is produced, modified, or used by a process.

Workflows
A workflow is a sequence of activities that produces a result of observable value.

In RUP all workers and activities are divided into six core engineering workflows and three core supporting workflows as shown in fig 10.

Business Modeling

In this discipline the scope of the system and its business context is modeled in business use cases. Common modeling activities include a context model, and a business process model. Often these activities are modeled in data-flow diagram or activity diagrams. The RUP provides common language for both business engineering community and software engineering community.

Requirements

In this core discipline functional and non-functional requirements are engineered, which includes identification, modeling and documenting requirements. Actors and Use cases are identified representing the users and system behavior. The use case description shows how the actors interact step by step with system and what the system does. Non-functional requirements are described in supplementary specifications.

Analysis and Design

The goal of this discipline is to show how system will be realized in the construction phase. The design model and optionally an analysis model are the outcomes of this discipline. The design model is used as a blue-print for source-code development. The Design model contains design packages and design subsystems with well defined interfaces. Finally the design activities are combined to build robust system architecture based on client requirements.

Implementation

In this discipline Classes and objects are implemented in terms of subsystems and components. Components are structured, tested and integrated into executable subsystems.

Testing

The process of testing is done in an iterative way throughout the project development, which allows finding defects as early as possible. The purpose of testing is to verify correct interaction between objects, to verify the implementation of all requirements and proper integration of all components. Testing is carried along three quality dimensions like reliability, functionality, application performance and system performance.

Deployment

The goal of the deployment discipline is to produce a successful product release. This includes activities: Like packaging the software, distributing the software, installing the software and providing help to users. All these deployment activities are centered on the transition phase.

Configuration and Change Management

This discipline describes how to control artifacts produced by many people who work on a common project. It also describes how to manage parallel development done at multiple sites and how to automate the build process.

Project Management

Software project management is a key discipline for managing risk, and successful delivery of products to customers and users in their prescribed way [15]. This process

is made easy by providing a framework for management risk and practical guidelines for planning, staffing, executing and monitoring projects in RUP.

Environment
The software development organization must provide the software development environment like processes and tools that are necessary to support the development team.

4.2 Use Case Modeling

As discussed in [18], Use case modeling is used primarily to capture the high level user- functional requirements of a system. A use case diagram shows a set of use cases and actors with their relationships UML modeling. A use case is a.
"Sequence of actions and variants between the system and the actors with their relationships". [10]
An actor is an external entity typically a user or another system, that interacts with a system by means of sending and receiving messages. An actor is depicted as a stick figure on a use case diagram. A primary actor triggers the system behavior in order to achieve a certain goal and a secondary actor interacts with the system but does not trigger the use-case. The system is depicted as a box and a use case is depicted as an ellipse inside the box. The use cases provide common understanding between developers, domain experts and end users [10]. Use case descriptions include the information related actor actions and system response in the form of scenarios. A scenario is "a specific sequence of actions that describe the behavior of a use-case at one instance". The main success scenario contains the sequence of actions that lead to successful completion of a goal. The sequences that may lead to the goal are described as alternate scenarios. The sequences that lead to failure in completing the goal are described as exceptional scenarios. A complete set of use cases specifies all the different ways to use the system. Use cases are useful in scoping, estimating, scheduling and validating the effort [18].

As mentioned in [18], three kinds of relationships have been defined between use cases: Dependency, Association and Generalization. The participation of an actor in a use case is known as "association", i.e. instances of the actor and instances of the use case communicate with each other. The generalization is defined as "a taxonomic relationship between a more general element and a more specific element that is fully consistent with the first element" [18]. Generalization is described between one or more actors, as a solid line from child to parent with open arrow head. Dependencies are use relationships between use cases. The other relationships between uses cases are the "extend" and "include". An extend relationship describes the extension of base use case to other use case; an instance of the other use case is included in the base use case provided the specified conditions are fulfilled. The extend relationship includes a condition for the extension and a reference to an extension point in the target use case. The include relationship provides explicit and unconditioned extensions to a use case; this means the behavior of included use case is inserted into base use case without any conditions. The below fig 11 shows the example use case diagram of an Arcade Game Maker Product line.

In which the actor game installer inherits all the properties of game player with two additional use cases. The ‘play game’ use case applies to all games. The include relationship between ‘play pong’ use case and ‘initialization’ use case shows some kind of dependency i.e. the ‘play pong’ is a base use case which uses the services of ‘initialization’ use case.

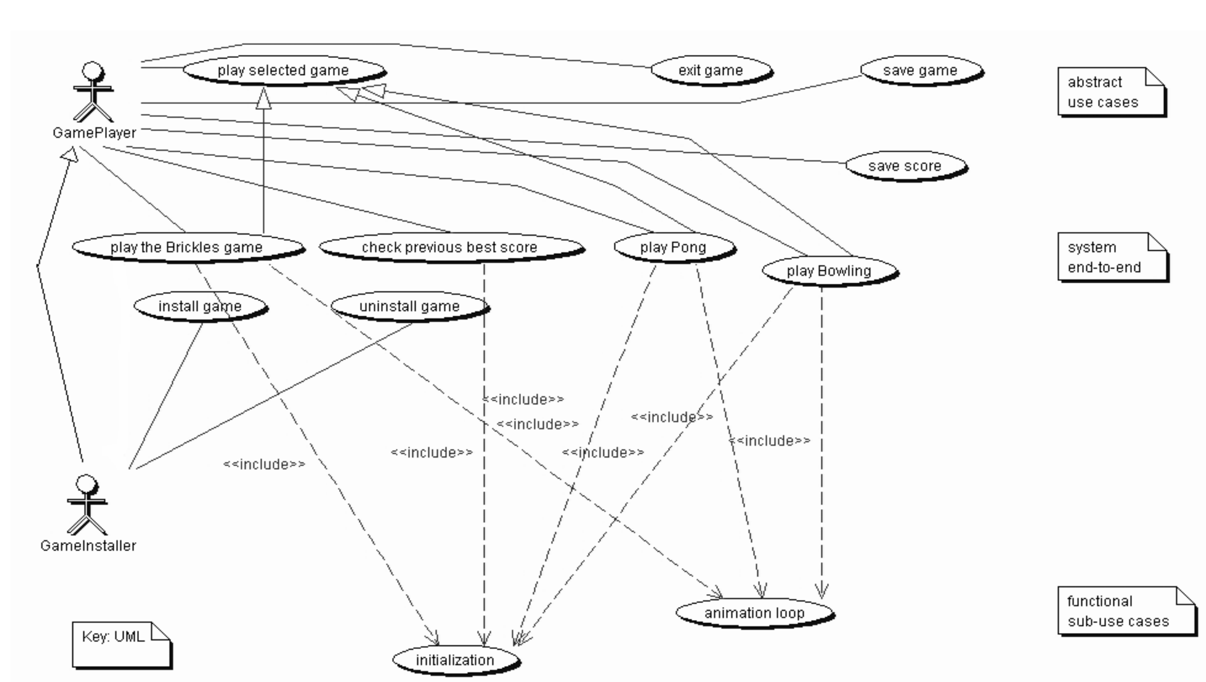


Fig 11: The system use-case model for Arcade Game Maker Product Line [16]

4.3 Rational Unified Process for Systems Engineering (RUP SE)

The RUP SE is an extension of RUP, developed specifically for addressing the needs of the systems engineering process. Systems engineering is “an interdisciplinary approach and means to enable the realization of successful systems” [5]. A system engineering process requires set of activities that are necessary to define the system architectural elements and their requirements. The RUP SE helps to unify the entire system design and development team and also improves the communication and collaboration between team members. The RUP-SE supports the large scale systems composed of software, hardware, workers and information components. In RUP SE system requirements are described two ways functional and non-functional, use case diagrams are used to describe the functional requirements and the other is non-functional requirements are scalability, performance, reliability and capacity etc. In order to derive the system requirements systems can be viewed in two different perspectives.

Black box perspective: In this the system is considered as whole.
White box perspective: The elements or parts that make up the system.

In RUP SE [21], the activity of deriving functional requirements for a system and its elements is called "Use-Case Flowdown". This flowdown can be applied at analysis level to identify sub-system level and to break sub-systems into further sub-systems. The outcomes of this activity are:

- A Use-case survey for subsystems
- A survey of hosted sub-system use case for localities
- A survey of realized subsystem use case for processes

The locality is an engineering viewpoint diagram, in which system is decomposed into elements and the localities are a collection of these elements that can host processing.

In this activity, first architecturally significant use-cases are identified. For each chosen use-case, the flow of events is developed to describe the interactions between the system actors and the system. The systems responses to the actions of actors are depicted as "blackbox" flow of events. Each black box step is shown along with performance requirements known as "blackbox budget requirements". Below fig12 shows the sample of flow of events for making a sale in retail store.

Step	Actor Action	Black Box	Black Box Budget Requirement
1	This use case begins when the **Clerk** pushes the New Sale button.	The **System** brings up new sale clerk and customers screens and enables the scanner.	Total response time is 0.5 second.
2	The **Clerk** scans the items and enters the quantity on the keyboard.	For each scanned item, the **System** displays the name and price.	Total response time is 0.5 second.
3	The **Clerk** pushes the Total button.	The **System** computes and displays on the screen the total of the item prices and the sales taxes.	Total response time is 0.5 second.
4	The **Clerk** swipes the credit card.	This use case ends when the **System** validates the credit card, and: If the credit card is valid, the **System** prints out a receipt, updates the inventory, sends the transaction to accounting and clears the terminal. If the credit card is not valid, the **System** returns a reject message	Total response time is 0.5 second.
		If the credit card is not valid, the **System** returns a reject message	Total response time is 30 seconds.

Fig 12: An example RUP SE Black box description [6]

In the second step, the Object Oriented Analysis and Design techniques are applied to identify the subsystem and the process models. In the third step the subsystem, locality and process models are used to revise the flowdown activity to define how the

analysis elements participate in carrying out the use cases. The specification of these design elements is called as “whitebox flow of events”. The first Blackbox step in the fig12 is decomposed into three Whitebox steps, as shown in below fig 13.

Step	Actor Action	Black Box	Black Box Budget Requirement	Subsystem White Box	White Box Budget Requirements
1	This use case begins when the **Clerk** pushes the New Sale button.	The **System** brings up new sale clerk and customers screens and enables the scanner.	Total response time is 0.5 second.	**The Point-of-Sale Interface** clears the transaction, brings up new sales screens, and requests that **Order Processing** start a sales list.	1/6 second.
				Order Processing starts a sales list.	1/6 second.
				Point-of-Sale Interface enables the scanner.	1/6 second.

Fig 13: An example RUP SE White box step description [6]

The purpose of subsystem whitebox steps is to illustrate how the subsystems collaborate to carry out each blackbox step. The final step in RUP SE is to determine the subsystem use cases. This process is carried out by sorting and organizing the whitebox steps associated with each subsystem according to the relation between them.

5 The PLUSS Approach

5.1 Introduction

The PLUSS [7] (Product Line Use case modeling for Systems and Software engineering), is a domain modeling approach proposed to address the systems requirements, and requirements reuse at system level especially in embedded software product line development. The idea of the PLUSS approach is to combine use cases and features into integrated model that provide a high level view of the system family and to reduce the risks associated with two separate models for use cases and features. The PLUSS approach uses features, use cases and use case realizations to identify and modeling the requirements. According to [7], the PLUSS approach is based on the work by Griss et al. on FeatuRSEB [8]. In PLUSS approach a feature model is used as a tool to visualize the variants in a system family use case model. In PLUSS one complete use case model is maintained for the whole system family to provide good

overview of dependencies within the model. The notations used in the PLUSS approach are easy to understand and easier to find the information about the domain.

5.2 The PLUSS Feature Modeling

As discussed in FODA [12], a feature is described as “A prominent or distinctive user-visible aspect, quality or characteristic of a system”. In PLUSS feature model, the top of the tree, root represents a concept or domain and reaming nodes represent the sub concepts. In PLUSS approach feature types are described as “Mandatory”, “Optional” and “Alternative” features as in FODA with relations “requires” and “excludes” among the features. In addition to that a new feature type is defined to represent the “atleast-one-out-of-many” relation called “Multiple adaptor features”, which is similar to FODA’s alternative features.
The mandatory features are included in all the systems built with in the family and optional features are may or may not be included in products and represent the variability with in a family of products. The alternative feature represents “exactly-one-out-of –many” selection among set of features and hence named as “Single adaptor feature”. The “requires” relation indicates the dependency of one feature on other to make a complete system. An “excludes” relations shows that both features cannot be included in the same system. Mandatory features are represented with filled black circle and a non-filled circle represents the optional feature. Single adaptor feature represented with ‘S’ and multiple adaptor feature represented with ‘M’, both surrounded by a circle. The example of PLUSS feature model is shown in fig 14.

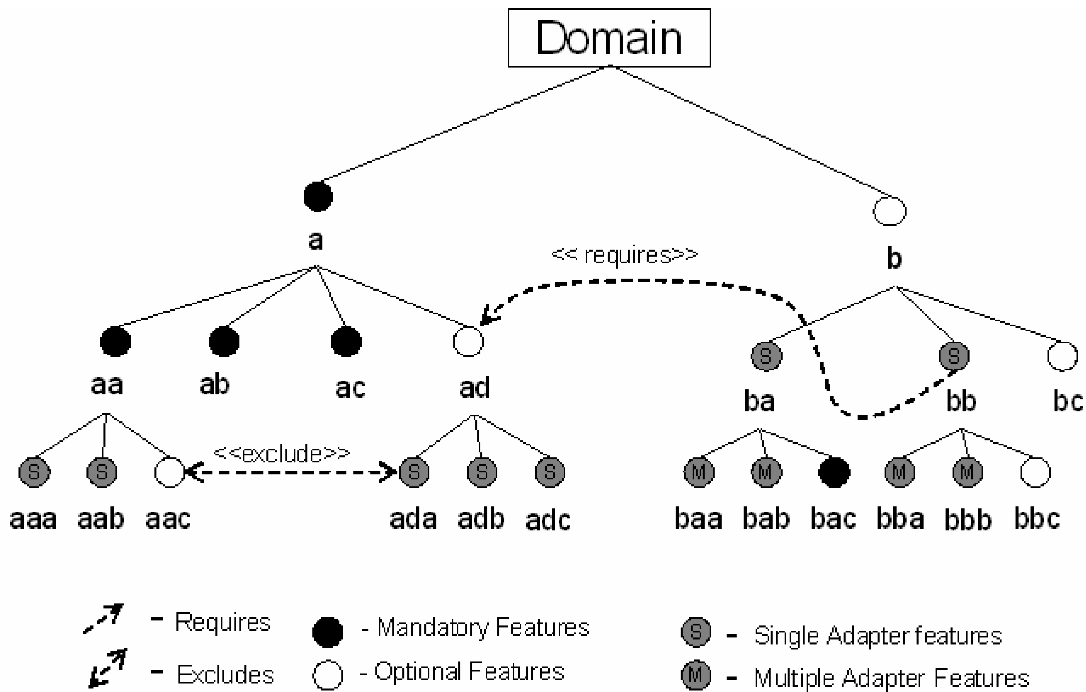


Fig 14: An example feature graph in the PLUSS notation [7]

To model the multiplicities several constructs are defined in the PLUSS approach as shown in fig 15. Multiplicity '*' represents the total number of features included in a set.

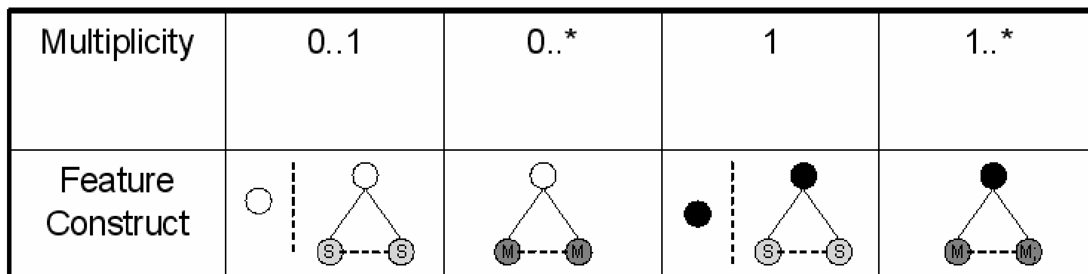


Fig 15: Feature constructs vs. Multiplicities in PLUSS [7]

5.3 The PLUSS Use Case Modeling

In the PLUSS approach the use case scenarios are described in natural language using the "Black Box Flow of Events" notation described in Rational Unified Process for Systems Engineering (RUP-SE) [21]. The tabular notation is shown in below fig 17.

Stop	Actor Action	Blackbox System Response	Blackbox Budget Requirements
1	This use case begins when the Actor…	The System…	It shall…
2	….	…	…
3	The use case ends when….	….	….

Fig 16: Blackbox flow of events used for describing use case scenarios [7]

This notation possesses two advantages over the traditional natural language scenario descriptions. That is it encourages the analysts to think about interfaces as it provides separate fields for actors action and system response and provide strong relation between non-functional requirements and blackbox budget requirements. The use case realizations are described using RUP-SE "White Box Flow of Events" as shown in fig 17. The use case realizations help the system designers to analyze how a use case is realized in terms of designing elements. The use case modeling notations of the PLUSS approach is easy to write and provides a good overview over functional and non-functional requirements of a system and software engineering.

Whitebox Action	Whitebox Budget requirements
Design Element_1…	It shall….
Design Element_2…	……
Design Element_3…	….
….	….
…	….

Fig 17: Whitebox flow of events for describing use case realizations [7]

A change case creates an impact on traceability to the variants if a change case implemented. A change case is basically a use case that specifies anticipated changes to a system. In PLUSS approach a change cases are used to differentiate between present and future views of a system

5.4 The PLUSS approach to Modeling Variants in Use Case Models

The variability management is a key concept of maintaining one complete and common use case model for the whole product family. In PLUSS approach four types of variants are identified and modeled. In PLUSS approach features and use cases are mapped as "many-to-many" relationship so that one feature model contains several user cases and one use case contains several features. A meta-model is proposed in the PLUSS approach used to integrate features, use cases and use case realization and also describes how use cases, use case scenarios and use case scenarios steps are included with features selections. The PLUSS meta-model is shown in the below fig 18.

Fig 18: The PLUSS Meta-model [7]

5.5 The PLUSS Notation for Describing Variants in Use case Specifications

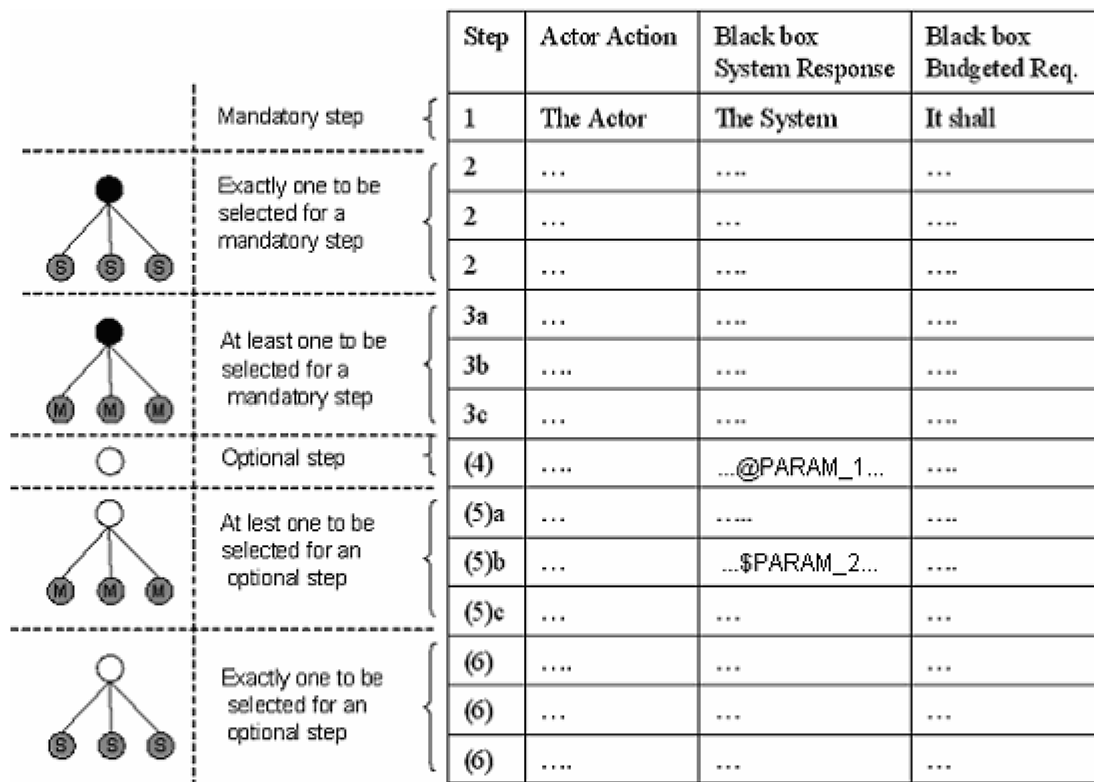
In PLUSS approach, the variants in Use case Specification are described in step identifier flow [7].

1. A step with number 1, identified as mandatory step in the scenario.
2. A number of alternative steps for one mandatory step identifies as several steps with same number. These steps must be related to a set of single adaptor features with a mandatory parent in the feature model, but exactly one step must be selected for a mandatory step.
3. Several steps identified with same number and consecutive letter are related to a set of multiple adaptor features with a mandatory parent of step 1. At least

one must be selected out of several multiple adapter features for a mandatory parent step.

4. A step identified by a number within the parenthesis as optional step in the scenario.
5. Several steps with same number and consecutive letter are identified as number of alternatives for one optional step in the scenario out of which at least one step must be selected. These steps are related to set of multiple adaptor features for one optional feature in the feature model.
6. Several steps identified with same number within parenthesis with a number of mutually exclusive alternatives for one optional step in the scenario, in which exactly one must be selected. These steps must be related to set of single adaptor features with an optional parent feature in the feature model.

In PLUSS the global and local parameters are denoted with '@' and '$' respectively. The scope of a global parameter is the whole domain model; where as the scope of local parameter is defined within the use case. The following fig 19 shows the example use case scenario description with variants.



Step	Actor Action	Black box System Response	Black box Budgeted Req.
1	The Actor	The System	It shall
2	…	….	…
2	…	…	….
2	…	….	….
3a	…	….	….
3b	….	….	….
3c	…	….	….
(4)	….	...@PARAM_1...	….
(5)a	…	…..	….
(5)b	…	...$PARAM_2...	….
(5)c	…	…	…
(6)	….	…	…
(6)	…	…	…
(6)	….	…	…

Fig 19: The PLUSS notation for describing variants in use case scenarios [7]

6. PLUSS Evaluation Results

As a result of the evaluation process of the PLUSS approach on Arcade Game Maker Product Line example, a feature model was developed as shown in appendix 1. The feature model and use cases of Arcade Game Maker example are described to provide

the traceability to the variants in the domain, as shown in appendix 2. The blackbox flow of events for use cases and change cases of Arcade Game Maker example are shown in appendix 3. The main purpose of this evaluation was documenting the problems associated with the PLUSS approach. The evaluation criteria are defined with the following five evaluation constraints and compared with original documentation available for Arcade Game Maker Product Line [16] the following grades as shown in below fig 20. During this evaluation process I found difficulties in identifying the feature types and its dependency, because I don't have the domain knowledge and product line designing experience. This evaluation is carried out with reference from the original documents of Arcade Game Maker Product line that are available at [16]. The documents of the PLUSS approach give detailed information about each element that should be taken care on designing phase.

	Evaluation Criteria	**Grades**
1	Ease of Learning	4
2	Richness of Notation	5
3	Tool Support	3
4	Needed modeling effort	4
5	Usefulness for Understanding the Domain	4

Fig 20: The evaluation criteria

The grades are given with reference of some standard modeling approach as shown in fig 21.

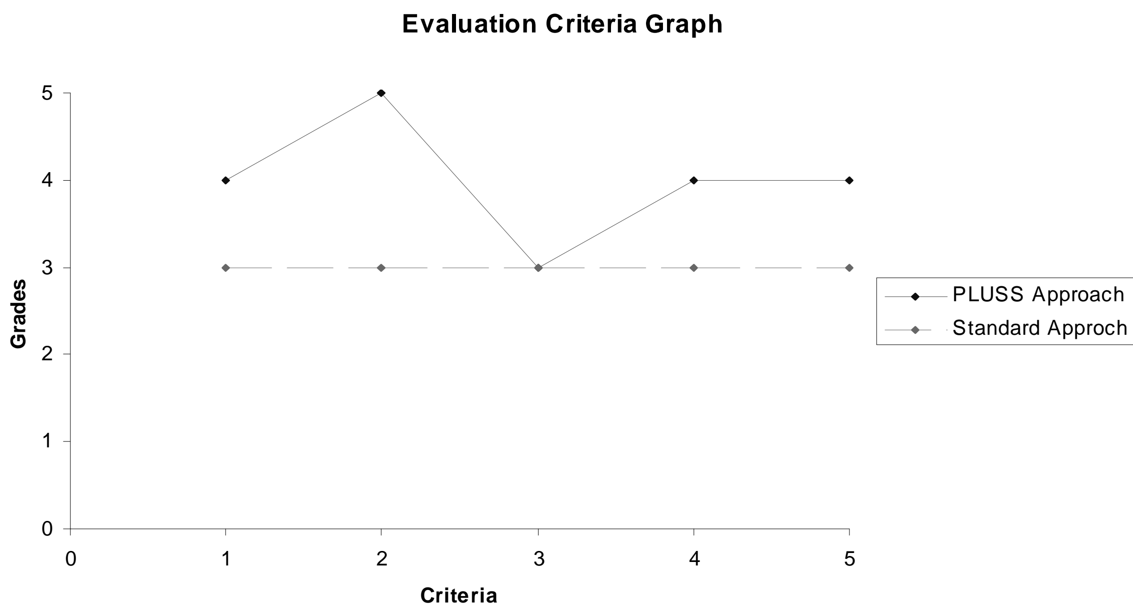


Fig 21: The PLUSS approach evaluation graph

Ease of Learning

The basic knowledge of software product lines, feature modeling and RUP-SE made it easy for me to learn the concepts of the PLUSS approach. The feature modeling and Use case modeling in the PLUSS approach is similar to the previous approaches with

some in-depth study. The working knowledge of PLUSS meta-model needs some domain knowledge to identify the variants in use case models.

Richness of Notation

The notations used in the PLUSS approach are easy to understand as one complete common use case model is maintained with integrated features of Arcade Game Maker Product Line. The feature types of the PLUSS approach made easy to derive the common and variable features. The relations 'exclude' and 'requires' describe the dependency among features. The black box flow of events and white box flow of events are used to describe use case scenarios and use case realizations. The notations address the better variant behavior along with commonality than any other approach and these notations can be easy to understand even without the software engineering knowledge.

Tool Support

As I don't have access to the commercial tools, I used some available tools in MS-Office suit. I am successfully drawn the feature model for Arcade Game Maker example. The problems associated with these tools are generating reports and tracing variants between models is not automated.

Needed Modeling Effort

I am successfully did the modeling of Arcade Game Maker product line example even without much domain knowledge and experience in modeling software product line architecture. But a person with proper domain modeling knowledge can easily use the PLUSS approach, when compared to some standard approach the PLUSS approach needs less effort.

Usefulness for Understanding the Domain

The documents of the PLUSS domain modeling approach provide the detailed information about the domain of Arcade Game Maker product line. These documents are written in natural language so that even a person without software knowledge can understand. One common use case model provides the total view of Arcade Game Maker and its domain.

7. Conclusion

To gain the benefits of a software product line strategy the software development organizations must select similar products and concentrate on their future requirements. A good evaluated approach for modeling the products under product line architecture plays a vital role. An approach said to be a good one when it can identify and derive all the elements that are needed for best product line architecture like use case scenarios, use case realizations, types of features, relationship among those features, predictable future requirements, traceability among variants and various parameters of a domain.

The PLUSS approach address several of those elements that are needed in the designing phase. The resulting models of Arcade Game Maker example provided good overview and easy way to find the information about the Arcade Game Maker domain. The specifications of use cases and change cases provide the exact and clear information to the designers to make good system architecture. The common model of the PLUSS approach for various products in the domain save the time and cost to the

company. The PLUSS approach is good tool for early cost estimations and provides high levels of reuse. Even though the PLUSS approach may have some drawbacks, but I couldn't find any may be because the domain of Arcade Game Maker is smaller and the games are not yet implemented commercially. Hence I would suggest the PLUSS approach as good domain modeling tool for only when use cases are used as use case driven approach product line development.

8. References

1. Bosch, J.: Design and use of Software Architectures, Addison -Wesley (2000)

2. Bosch, J., J. V. Gurp, and M, Svehnberg: On the Notation of Variability in Software Product Lines, In Proceedings of working IEEE/IFIP Conference on Software Architecture (WICSA'01), 2001 http://www.cs.rug.nl/~bosch/papers/SPLVariability.pdf (2005-02-21, 20:44)

3. Clements. P, Northrop. L.; Software Product Lines, Practices and Patterns, Addison-Wesley, (2002)

4. Czarnecki, K., Eisenecker, U.W.: Generative Programming. Addison -Wesley, 2000

5. Canter. M., Principal Engineer, Rational Brand Services, IBM Software Group Introducing RUP-SE version 2.0, Available at: (2005-03-23, 14:20) http://www-106.ibm.com/developerworks/rational/library/content/RationalEdge/aug03/f_rupse_mc.pdf

6. Eriksson M., Bösler J., Borg K.: Marrying Features and Use Cases for Product Line Requirements Modeling of Embedded Systems, Proceedings of Fourth Conference on Software Engineering Research and Practice in Sweden (SERPS'04), Available at: (2004-11-10, 15:30) http://www.cs.umu.se/~magnuse/papers/ErikssonSERPS04.pdf

7. Eriksson M., Bösler J., Borg K.: The PLUSS Approach- Domain Modeling with Features, Use cases and Use case Realizations. To appear in the proceedings of the International Software Product Line Conference (SPLC'05) 2005.

8. Griss. M. L., J. Favaro, M. D' Alessandro, Integrating Feature Modeling with RSEB. In Proceedings of the Fifth International Conference on Software Reuse, Vancouver, BC, Canada, 1998 (76-85)

9. Heymans P., Trigaux. J. C.: Modeling Variability Requirements in Software Product Lines: a comparative survey. (2003)

10. Jacobson, I., Rumbaugh, J., Booch, G.: The Unified Modeling Language User Guide, Addison – Wesley, 1998

11. Krueger C. W.: PhD, CEO, BigLever Software, Available at: http://www.softwareproductlines.com/benefits/benefits.html (2005-02-03, 10:45)

12. Kang K. Cohen S., Hess J., Novak W., Peterson A.: Feature Oriented Domain Analysis (FODA) feasibility study, Technical report CMU/SEI -90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA (1990) (2005-01-27, 18.30)

13. Kang K., Kim, S., Lee, J., Kim, K., Shin E. and Huh, M.: FORM: A Feature-Oriented reuse Method with Domain –Specific Reference Architectures, Annals of Software Engineering, 5 (1998).

14. Kang K. C., K. Lee, J. Lee: Concepts and Guidelines of Feature Modeling for Product Line Software Engineering, Phong University of Science and Technology, 2002

15. Mohr J.: Professor of Computing Science, University of Alberta, Alberta, Canada. http://www.augustana.ab.ca/~mohrj/courses/2000.winter/csc220/papers/rup_best_practices/rup_bestpractices.pdf (2005-03-20, 16:47)

16. McGregor. J. D., the Arcade Game Maker Product Line Example, Available at http://www.cs.clemson.edu/%7Ejohnmc/productLines/example/frontPage.htm (2004-11-10, 08:10)

17. Northrop L. M.: SEI, A Framework for Product Line Practice, Available at: http://www.sei.cmu.edu/productlines/framework.html (2004-11-10, 12:30)

18. OMG: Unified Modeling Language version 2.0, Available at: http://www.uml.org (2005-04-18, 14:15)

19. Reibisch M.: Towards a More Precise Definition of Feature Models, Technical University Ilmenau, Germany. (2005-02-10, 09.23) http://www.theoinf.tu-ilmenau.de/~riebisch/publ/06-riebisch.pdf

20. Reibisch M., Böllert K., Streitferdt D., Philippow I.: Extending Feature Diagrams with UML Multiplicities, In Proceedings of the Sixth Conference on Integrated Design and Process Technology (IDPT), Pasadena, CA, June 2002 http://www.theoinf.tu-ilmenau.de/~streitdf/TheHome/own/data/IDPT2002-paper.pdf (2005-02-16, 20:01)

21. Rational Software: The Rational Unified Process for Systems Engineering Whitepaper, ver.1.1, 2003, Available at: (2005-02-20, 15:38) http://www.rational.com/media/whitepapers/TP165.pdf

22. Schlee M.: Software Engineer, DFA, Generative Programming of Graphical User Interfaces, Thomson grass valley, Brunnenweg 9, Weiterstadt, Germany. http://www.care-t.com/events/mbui-workshop2004/papers/P02.Schlee.pdf (2005-02-16, 20:24)

Appendix 1: Arcade Game Feature Diagram

Appendix 2: Arcade Game Feature Model Description

Appendix 3: Arcade Game Use-Case Specifications