

Use Case Description of Requirements for Product Lines

A. Bertolino[^], A. Fantechi^{*}, S. Gnesi[^], G. Lami[^], A. Maccari⁺

^{*}Dip. di Sistemi e Informatica - Università di Firenze - Italy

[^]Istituto di Elaborazione della Informazione - C.N.R. - Area della Ricerca C.N.R. - Pisa - Italy

⁺ Nokia Research Center, Software Architecture Group - Finland

Abstract

Capturing the variations characterizing the set of products belonging to a product line is a key issue for the requirements engineering of this development philosophy. This paper describes ways to extend the well-known Use Case formalism in order to make possible the representation of these variations, in the perspective to make them suitable for an automatic analysis.

1. Introduction

The development of industrial software systems may benefit from the adoption of a development cycle based on the so called *system-families* or *product lines* approach [2,5]. This approach aims at lowering production costs when the products share the overall architecture and conception, but differ with respect to particular characteristics. These variations represent the customisation of the product with respect to the other members of a family of systems. The production process is therefore organized in product lines with the aim of maximizing the commonalities of the product family and minimizing the cost of variations.

In the first stage of a software project, called usually *requirements elicitation*, the knowledge of the system under construction is acquired. Inside a product line, both problems of capturing requirements common to all members of the product family, on one side, and of specializing the general product family requirements into those ones of a single product, on the other side, have to be addressed.

To deal with these problems, a close look to the nature of the relations between family and product requirements should be given: in these relations the concepts of parameterisation, specialization and generalization can play a major role.

On the other hand, the problems are not easily formalized, due to the nature of requirements, which are often given in a natural language prose. Without sacrificing the immediateness of the natural language, at least a notation that adds structure to the requirements should be adopted.

Use Cases are a powerful tool to capture functional requirements for software systems. They allow for structuring the requirements documents according to user goals and provide a means to specify the interactions between a certain software system and its environment. In his book on how to write Use Cases [1], Alistair Cockburn presents an effective technique for specifying the interactions between a software system and its environment. The technique is based on natural language specifications (i.e., phrases in plain English language) for scenarios and extensions. This makes requirements documents easy to understand and communicate even to non-technical people.

The purpose of this paper is to present how to extend Use Cases in the direction of addressing Product Line requirements, including some specific constructs to deal with variability.

This paper is structured as follows: in section 2 we describe the principal characteristics of Use Cases, in section 3 we discuss the impact of variability on the Product Line requirements. In section 4, possible extensions to Use Cases to face variability are discussed; finally in section 5, conclusions and future research directions are presented.

2. Background: Use Cases

A Use Case describes the interaction (triggered by an external actor in order to achieve a goal) between a system and its environment. A Use Case defines a goal-oriented set of interactions between external actors and the system under consideration. The term *actor* is used to describe the person or system that has a goal against the system under discussion. A primary actor triggers the system behaviour in order to achieve a certain goal. A secondary actor interacts with the system but does not trigger the Use Case.

A Use Case is completed successfully when that goal is satisfied. Use Case descriptions also include possible extensions to this sequence, e.g., alternative sequences that may also satisfy the goal, as well as sequences

that may lead to failure in completing the service in case of exceptional behaviour, error handling, etc.. The system is treated as a "black box", thus, Use Cases capture *who* (actor) does *what* (interaction) with the system, for what *purpose* (goal), without dealing with system internals. A complete set of Use Cases specifies all the different ways to use the system, and therefore defines the whole required behaviour of the system. Generally, Use Case steps are written in an easy-to-understand, structured narrative using the vocabulary of the domain. A scenario is an instance of a Use Case, and represents a single path through the Use Case. Thus, there exists a scenario for the main flow through the Use Case, and as many other scenarios as the possible variations of flow through the Use Case (e.g., triggered by options, error conditions, security breaches, etc.). Scenarios may also be depicted in a graphical form using UML Sequence Diagrams.

Figure 1 shows the template of the Cockburn's Use Case taken from [1]. In this textual notation, the main flow is expressed, in the "Description" section, by an indexed sequence of natural language sentences, describing a sequence of actions of the system. Variations are expressed (in the "Extensions" section) as alternatives to the main flow, linked by their index to the point of the main flow from which they branch as a variation.

USE CASE #	< the name is the goal as a short active verb phrase >	
Goal in Context	<a longer statement of the goal in context if needed >	
Scope & Level	<what system is being considered black box under design > <one of: Summary, Primary Task, Sub-function >	
Preconditions	<what we expect is already the state of the world >	
Success End Condition	<the state of the world upon successful completion >	
Failed End Condition	<the state of the world if goal abandoned >	
Primary, Secondary Actors	<a role name or description for the primary actor >. <other systems relied upon to accomplish use case >	
Trigger	<the action upon the system that starts the use case >	
Description	Step	Action
	1	<put here the steps of the scenario from trigger to goal delivery, and any cleanup after >
	2	<... >
	3	
Extensions	Step	Branching Action
	1a	<condition causing branching > : <action or name of sub-use case >
Sub-Variations		Branching Action
	1	<list of variations >

Figure 1. Use Case template

3. Variability in PF Requirements

Following the System Family Engineering Process Reference Model defined in the CAFÉ project [6], and shown in Figure 2, product family development is characterized by two processes: Application engineering and Domain engineering. Domain engineering is the process aiming at developing the general concept of a family together with all the assets which are common to the products of the family, whereas Application engineering is intended the process aiming at designing a specific product. During Application engineering a customer specific application will be defined. However, differently from the usual single product development, the definition process of the customer specific application is not only influenced by the requirements of the customer but also by the capabilities of the product family.

This diagram shows that it is possible to move from the family level (by means of the system family engineering activity) to the product level and vice versa (by means of the system family reverse engineering activity).

Going upwards, applications are developed considering the capabilities of the product family specialising, extending and adding family requirements. Consequently, software product families need more sophisticated requirement processing and requirements should deal with some variability notion.

In particular, product family requirements can be considered in general as composed of a constant and a variable part. The constant part includes all those requirements dealing with features or functionalities common to all the products belonging to the family and that, for this reason, do not need to be modified. The variable part represents those functionalities that can be changed to differentiate a product from another.

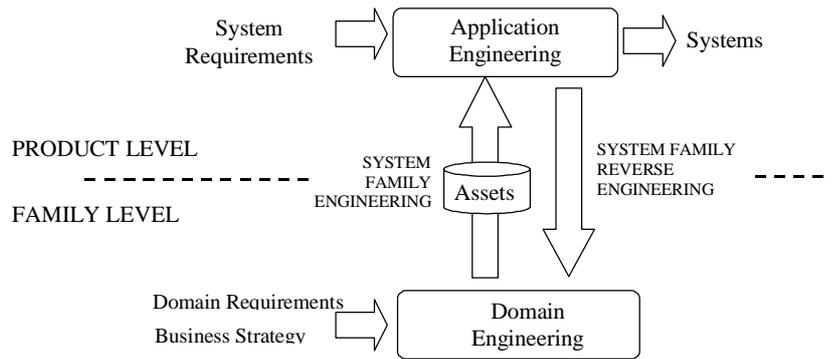


Figure 2. The CAFÉ-PRM reference framework

Following the above process we can see variability from two different perspectives: the first is the product perspective where each variability has to be considered as an aspect to be instantiated. Whereas, from the family perspective a variability is a goal to be reached by abstracting all the instances related to the existing products belonging to a product line.

It is possible to move down from the family level to the product level by an instantiation process and on the contrary from the product level up to the family level by an abstraction process. In these two different processes the main objects to pay attention on are variations.

We therefore aim at identifying needed extensions to express variability during requirements engineering.

Natural Language (NL) processing techniques, such as those proposed in [4] for evaluating requirement documents, can be fruitfully employed to this aim. Usually these techniques are used to detect and then remove ambiguity and vagueness in a requirements document because in traditional development process it is an undesirable side effect of the use of NL; in the context of Product Lines, on the contrary, we want to use them to identify automatically the necessary generic parts in order to specialise them into the specific product features (see the following examples):

Example 1

Req 34: *the system shall simulate different vehicles*

Different is a vague word, detected by the tools proposed in [4]

In a family description *different* could be considered as pointing at a variability to be specialized in the derived products:

Req A.34: *the system shall simulate cars*

Req B.34: *the system shall simulate trains*

Example 2

Req 217: *the system shall be such that the mission can be pursued, possibly without performance degradation*

Possibly is a word indicating optionality.

In a family description *possibly* could be considered as pointing at an optionality differentiating among products:

Req A-217: *the system shall be such that the mission can be pursued, without performance degradation*

Req B-217: *the system shall be such that the mission can be pursued (performance degradation is admitted)*

Notice that this is not the only interpretation of the optionality: the family requirement can be read also as imposing that any product will make its best effort to avoid performance degradation, if this is possible under adverse conditions. Hence, this kind of linguistic analysis can be useful to point out *potential* variability.

4. Extensions of Use Cases for Product Lines

When adopting Use Cases description of requirements for Product Families, variations can be addressed in Use Cases by adopting two complementary approaches:

1. Structuring the Use Case requirements as having two levels: the family level and the product level. In this way product-related Use Cases should be derived from the family-related Use Cases by an instantiation process;

- Incorporation of the application level Use Cases into the domain level Use Case, in this way both the product and the family level requirements will stand at the same level into the same, all inclusive, Use Cases document.

4.1 Two levels Use Cases

The solution for capturing variability we discuss in this section is based on the inclusion of *tags* into the scenarios (both main scenario and extensions) that identify and specify variations. The tags can be of three kinds: Optional, Alternative, Parametric.

When a product is derived from the family, the variable parts must be instantiated in different ways according to the type of variability:

- Alternative components: they express the possibility to instantiate the requirement by selecting an instance among a predefined set of possible choices, each of them depending on the occurrence of a condition;
- Parametric components: their instantiation is connected to the actual value of a parameter in the requirements for the specific product;
- Optional components: their instantiation can be done by selecting indifferently among a set of values which are optional features for a product instantiation.

The instantiation of these types of variability will lead to a set of different product-related instantiated Use Cases. Figure 3 expresses this situation as a UML class diagram.

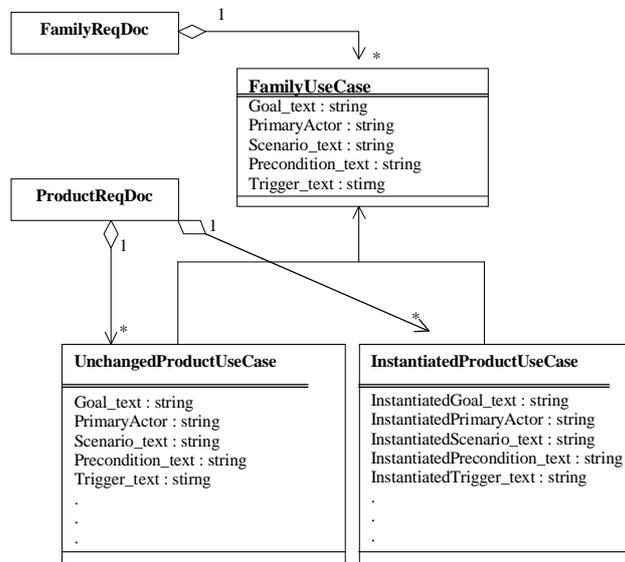


Figure 3. A UML model of two levels Use Cases Document

Example 3.

Primary Actor: the {[V0] 33-serie} mobile phone (the system)

Goal: play a game on a {[V0] 33-serie} mobile phone and record score

Preconditions: the function GAME has been selected from the main MENU

Main Success Scenario:

- The system displays the list of the {[V1] available} games
- The user select a game
- The system displays the logo of the selected game
- The user selects the difficulty level by following the {[V2] appropriate} procedure and press YES
- The system starts the game and plays it until it goes over
- The user records the score achieved and {[V3] possibly} send the score to Club Nokia via WAP
- The system displays the list of the {[V1] available} games
- The user presses NO

V0: alternative

V0: 1. Nokia 3310 model

```

2. Nokia 3330 model
V1: optional
    if V0=1 then game1 or game2
    else if V0=2 then game1 or game2 or game3
V2: parametric
    if V0=1 then procedure-A:
        - press Select
        - scroll to Options and press YES
        - scroll to Difficulty Level and press YES
        - select the desired difficulty level, press YES
    else if V0=2 then procedure-B:
        - press Select
        - scroll to Level and press YES
        - select the desired difficulty level, press YES
V3: parametric
    if V0=1 then function not available
    else if V0=2 then function available

```

4.2 One level Use Cases

The second way to manage variations in Use Cases is to include all the alternatives that can occur when a product will be considered in each Use Case. In this way the canonical structure of the Use Case should be modified in order to be suitable to include all the particular features that the whole product family set of instances can have. Figure 4 shows, again by a UML class diagram, that each Use Case includes all the possible alternatives due to the specialization of the generic Use Case.

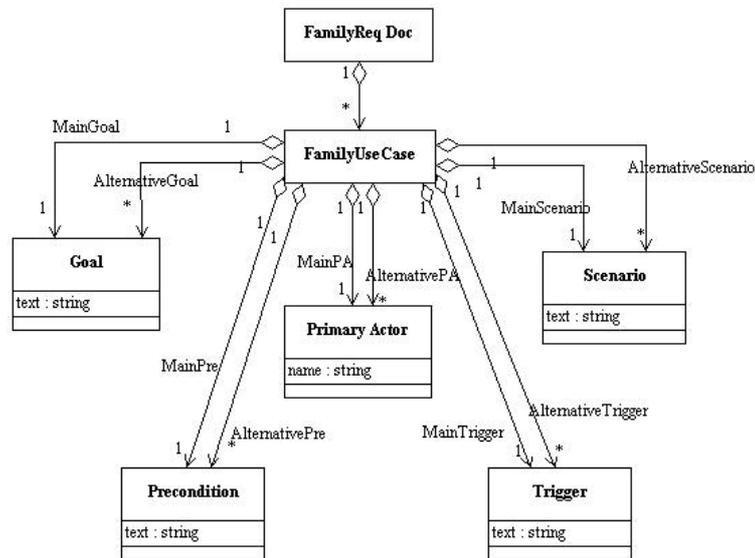


Figure 4. A UML model of one level Use Cases Document

Example 4.

Primary Actor: The 3310 - 3330 mobile phone

Goal: play a game on a 3310 - 3330 mobile phone and record score

Preconditions: the function GAME has been selected from the main MENU

Main Success Scenario:

- 3310 model: The system displays game1 or game2
- 3330 model: The system displays game1 or game2 or game3
- The user select a game
- The system displays the logo of the selected game
- The user selects the difficulty level by following the procedure below:
 - 3310 model:
 - press Select
 - scroll to Level and press YES
 - select the desired difficulty level and press YES
 - 3330 model:
 - press Select
 - scroll to Options and press YES
 - scroll to Difficulty Level and press YES
 - select the desired difficulty level and press YES
- The system starts the game and plays it until it goes over

- The user records the score achieved
- 3330 model: send the score to Club Nokia via WAP
- 3310 model: The system displays game1 or game2
- 3330 model: The system displays game1 or game2 or game3
- The user presses NO

4.3 Discussion

The extensions presented in the previous sections reflect different perspectives from which variability can be seen. The first approach considers the variations implicitly enclosed into the components of the Use Cases. The variations are then represented by tags that indicate those parts of the family requirements needing to be instantiated for a specific product in a product-related document. On the contrary, the other extension explicitly includes all the possible variations into a unique, all inclusive, document.

The tagged extension, because its dynamic structure, allows modifications to be faced more easily in terms of new functions or products in the product line. The flat, all inclusive approach is more static and, since it provides plenty of information at the same level.

By considering the nature of the two approaches, the flat representation of the use cases seems suitable for functionalities that are already mature and almost stable; while the tagged representation is more suitable when the functionalities related to the use case are still in the way to be precisely defined.

In reference to the CAFÉ-PRM reference framework, these considerations imply that the tagged extension is more useful during Application Engineering, in which variations are introduced on top of already existing products. During Domain Engineering the comprehensiveness of the flat extension allows to develop the general concept of a family.

The trade-offs between the two approaches have anyway to be better investigated. In fact, the two approaches may co-exist into the same Use Case document, where some parts may be expressed by using the tagged way and some other by using the flat way.

5. Conclusions and Future Works

In this paper we presented two possible ways to define extensions on the canonical Use Case structure in order to make them suitable to manage the variations of the product line requirements. The impact of these two approaches on the structure of the use cases and the consequent extensions to be made have been discussed in the paper.

The proposed Use Case extensions seem to be suitable to perform linguistic analysis to find ambiguity, inconsistency and incompleteness defects by means of automatic tools. Next work on this argument will deal on the application of existing linguistic techniques [3], [4] for evaluation and defects detection in use cases-based product line requirements. Furthermore, we intend to explore the respective pros and cons of the two approaches in practice.

6. References

- [1] A. Cockburn. Writing Effective Use Cases, Addison-Wesley, 2000
- [2] P. Clements, L.Northrop. Software Product Lines: Practice and Patterns, SEI Series in Software Engineering Addison Wesley, 2001.
- [3] F.Fabbrini, M.Fusani, S.Gnesi, G.Lami. The Linguistic Approach to the Natural Language Requirements Quality: Benefits of the use of an Automatic Tool, 26th Annual IEEE Computer Society - NASA GSFC Software Engineering Workshop, Greenbelt, MA, November 27-29 2001.
- [4] A.Fantechi, S.Gnesi, G.Lami, A.Maccari. Linguistic Techniques for Use Cases Analysis, Proceedings of the IEEE Joint International Requirements Engineering Conference - RE02. Essen, Germany, September 9 -13 2002.
- [5] M. Jazayeri, A. Ran, F. van der Linden. Software Architecture for Product Families: Principles and Practice, Publishers: Addison-Wesley, Reading, Mass. and London, 1998.
- [6] F. van der Linden Software Product Families in Europe: The ESAPS & Café Projects, IEEE Software July/August 2002