Fraunhofer Institut
Experimentelles
Software Engineering

Modeling Variability with Use Cases

Authors:
Isabel John
Dirk Muthig

In part supported by the ITEA Project
EMPRESS Project Nr. 00103

IESE-Report No. 063.02/E
Version 1.0
November 7, 2002

A publication by Fraunhofer IESE

Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft.
The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competetive market position.

Fraunhofer IESE is directed by
Prof. Dr. Dieter Rombach
Sauerwiesen 6
D-67661 Kaiserslautern

Abstract

Use cases are used for single system requirements engineering to capture requirements from an external point of view. When utilizing use cases for product line modeling they cannot be used as is but they have to be extended with a variability mechanism. Stereotypes can be used as this variability mechanism for use case diagrams and tags can be used for textual use cases. In this report we describe how to tailor use cases for product line modeling, describe in which situations the approach can be applied and illustrate the use case approach by an example.

The work described herein is based on two publications by Isabel John and Dirk Muthig "Product Line Modeling with Generic Use Cases" [12] published at SPLC2 and "Tailoring Use Cases for Product Line Modeling" [13] published at RE'02.

Keywords: Product Line Engineering, Requirements Engineering; Domain Modeling, Use Cases, Variability

Table of Contents

1 Introduction

During the last decade reuse has been recognized as a key factor for improving software development efficiency. Product line engineering is a reuse approach providing methods to plan, control, and improve a reuse infrastructure for developing a family of similar products. The goal of product line approaches, such as PuLSE™ [3] is to achieve a planned domain-specific reuse by building a family of applications rather than developing products separately. Distinct from single-system software development, there are two main life-cycle phases: domain and application engineering. Domain Engineering constructs the reuse infrastructure, which is then used by application engineering to build the required products.

During the domain-analysis phase of domain engineering the common and varying requirements of the planned set of products are captured. Use cases are a widely accepted means to support domain understanding, find, and document user requirements but there is no generally accepted formalism that integrates variability modeling with use cases in order to do product line modeling.Expressing variability in the use cases has benefits in the following ways:

- Seeing variability in the use-cases helps all involved roles in establishing a variability and product line mindset and in getting a better domain understanding
- Explicit variability in use cases supports the instantiation and derivation of exact models in application engineering
- In market development which is not customer oriented, variable use cases are a good means of communicating the possibilities of the possible products between marketing and requirements engineers and product line engineers

In order to be suitable for product line modeling, commonality and variability has to be integrated and described in use-case diagrams and textual use-case descriptions. Furthermore decision modeling has to be supported. We illustrate the adaptations we made with an example, a "cruise control system" that is a part of the automotive domain. A cruise control system supports the driver in keeping a constant velocity and does real time monitoring and control of the cars speed. It exists in variants that have or have not a distance regulator, which controls the distance between the car to the car(s) ahead and behind.

In this report we describe a method tho extend use case diagrams and textual use cases with explicit commonality and variability. The work described herein is based on two publications by Isabel John and Dirk Muthig "Product Line Modeling with Generic Use Cases"[12] published at SPLC2 and "Tailoring Use Cases

for Product Line Modeling"[13] published at RE'02.
There are some approaches on how to extend use cases with variability and how to support reuse and genericity in use cases. Biddle et.al. [5] suggest to use patterns in use cases and to organize the use cases in a repository. However, they do not introduce variability. America and Wijnstra [1] describe how to use use cases in requirements engineering for product lines. Halmans and Pohl [9] show how application derifaction can be supported by product family specific use cases but they do not give a concrete notation. Gomaa [7] introduced in his method the stereotypes <<kernel>> and <<optional>> for use cases and other UML model elements for modeling families of systems. In his approach he focuses on the integration of features and use cases but does not say anything about how textual use cases should be represented. Jacobson et.al. [11] discuss how the text in use cases may involve variation points that can form the basis for a hierarchy of use cases from more abstract use cases to more specific ones. They introduce "variation points" (described as dots in use cases including a short description of the variation) into use case diagrams. They also use variation points in textual use cases (described as highlighted text in curly brackets) to describe different ways of performing actions within a use case. They do not say anything about how variant or generic use cases can be instantiated. Our approach is related to the KobrA approach [2], which introduces variation points in the UML including use cases and supports the instantiation of generic models.

2 Single System Use Cases

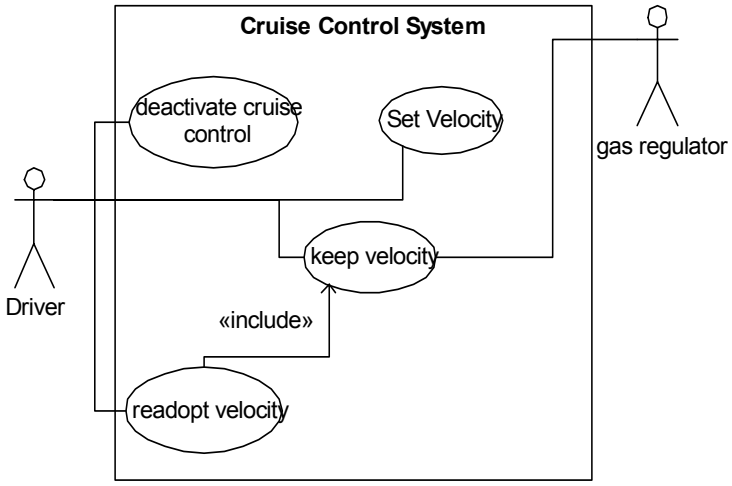
2.1 Requirements Engineering with Use Cases

Requirements Engineering for single systems is adressed in research and in practice since many years. Methods and Formalism which have been used for a long time are e.g. textual requirements, controlled languages, formal or semi-formal specification languages like SDL or scenarios. For some years, use cases [10] have been a means to understand, specify, and analyse user requirements that is rather often used. Use cases can document the requirements on a system from an outside or users point of view. Use cases cannot document all requirements, they normally do not deal in depth with non-functional requirements, with interfaces, and data formats. But use cases make it easier to understand what the user wants from the system and give a good means of communication about the system.

2.2 Use-Case Diagrams

A use case describes the system's behavior under various conditions. Use cases are used during the analysis phase to identify and partition system functionality. A use case describes the actions of an actor when following a certain task while interacting with the system to be described. A use case diagram includes the actors, the system, and the use cases themselves. The set of functionality of a given system is determined through the study of the functional requirements of each actor, expressed in the use cases in the form of common interactions. So a use case diagram in UML 1.4 consists of [15]: the system to be described, the use cases within the system, the actors outside the system and the relationships between actors and use-cases or in between use cases (associations, generalization, include, and extend). Associations denote the participation of an actor in a use case, a generalization relation means that there is a specialization of one use case to another. An extend relationship indicates that an instance of a use case may be augmented by the behavior specified by another use case and the include relationship indicates that an instance of a use case will contain the behavior of another use case. Figure 1 shows an example use case diagram for a cruise control system. The use cases described in this diagram are on the "Sub-function level" or "Underwater level" in Cockburns classification of Use case levels [6]. The driver activates the cruise control system by choosing "`set velocity`". He can also tell the system to "`keep velocity`" with help of the gas regulator if the velocity has already been set. He can also "`readopt velocity`" which will bring the car to the fixed speed (e.g. after braking) and then continue keeping the velocity.

Figure 1:
A Use Case Diagram for a Cruise Control System



2.3 Textual Use Cases

There is no standardized form for the content of a use case itself, the standard describes the graphical representation and the semantics of use case diagrams only. Use cases are fundamentally a text form although they can be written using flow charts, sequence charts or petri nets [6]. Use cases serve as a means of communication from one person to another, often among persons with no training in UML or software development. So writing use cases in simple text is usually a good choice. There is no general agreement on the attributes use cases should have and on the level of description of the use cases. Figure 2 shows an example of a textual use case in the cruise control domain which describes the use case "`keep velocity`". The template used is a modification of the template suggested by Alistair Cockburn (see [6]). The use case is described with its actors, the triggers, which means the actors that can activate the use cases. The input and output of the use case are described and the post conditions and a success guarantee (what the user wants from the use case) and a minimal guarantee (what should in any case not go wrong) are given. The main part of the use case is the main success scenario which describes what the use case actually does.

Figure 2:
A use case for "keep velocity" of the cruise control system

Use Case Name: keep velocity
Short Description: keep the actual velocity value over gas regulator
Actors: driver, gas regulator
Trigger: actor driver
Precondition: --
Input: starting signal, velocity value vtarget
Output: infinit
Postcondition: vactual = vtarget
Success guarantee: vactual = vtarget
Minimal guarantee: The car keeps driving
Main Sucess Scenario:

1.) the actor driver selects <keep velocity>
get vactual, vtarget (<Calculate Velocity>)
- compare vactual and vtarget
If vactual < vtarget : gas regulator increase velocity
- restart <keep velocity>
If vactual > vtarget : gas regulator decrease velocity
- restart <keep velocity>
else restart <keep velocity>

3 Tailoring Use Cases for Product Lines

In this section, the use case approach is extended to product families, that is, use cases do not longer describe the actions of an actor when following a certain task while interacting with a particular system only, but summarize and integrate use cases describing analogous tasks for different products in a family into combined artifacts, product-line use cases. We describe how and why we tailored single system use cases to capture variability and commonality and describe how a decision model of those use cases can be built. Before describing in Section 3.1 the required extensions to the two artifacts, use-case diagram and textual use-case descriptions, described above, the main product-line concepts are introduced.

3.1 Product Line Concepts

From an abstract point of view it is the concurrent consideration, planning, and comparison of similar systems that distinguishes product line engineering from single-system development. The intention is to systematically exploit common system characteristics and to share development and maintenance effort.

In order to do so, the common and the varying aspects of the systems must be considered throughout all life-cycle stages and integrated into a common infrastructure that is the main focus of maintenance activities. Commonalities and variabilities are equally important: commonalities define the skeleton of systems in the product line, variabilities bound the space of required and anticipated variations of the common skeleton. In Product Line Modeling, modeling commonalities and Variabilities is often done with feature modeling (either with FODA [14] or FeatuRSEB [8]). But feature models often describe a static view on the systems capabilities. In order to get a dynamic view on the system use cases are a good choice as a modeling approach that should be used in addition to feature models.

When the concepts of commonalities and variabilities are applied to use cases, these concepts produce use cases that have a common story that is valid for all members of a system family with variation points that explicitly capture which actions are optional or alternatives. Of course, a use case as a whole may be optional, as well as use cases under the same label may be realized totally different for some products. The common parts are modeled as all parts in a single-system context, to model variation, additional means are required. The variant use cases are instantiated during application engineering. The instantiation process is guided by a decision model, which captures the motivation and interde-

pendencies of variation points, and produces use case artifacts as used in a single-system context (see previous section). We will illustrate, how decision modeling and instantiation can be done with use cases. These use cases approach we describe is used within PuLSE CDA [4], the customizable domain analysis part of the PuLSE™ [3] product line framework. CDA is customizable which means, domain analysis is not always done with the same approach (like e.g. feature modeling) but customized to the context where the domain analysis approach is applied. Based on a set of fixed customization factors (c.f. [16]) we apply this use case approach in the following situation:

Domain characteristics: use cases can be applied in almost all domains. They can be useful for embedded or information systems. The domain should address user-level information (e.g have a user interface). The number of possible variation points in the domain should be low to medium (2 to 100) otherwise the use cases are not readable anymore

Information sources: Use cases can be derived from paper documents, experts or legacy systems. So they can be used with all information sources

Implementation characteristics: As use cases are part of UML use case models are encouraged if an Object oriented modeling and implementation approach is chosen. For clarification they can also be used with other implementation approaches

Integratable software artifacts: The use case approach is independent of existing software artifacts

Project context: A complete set of use cases and use case diagrams makes sense in larger projects with a staff size bigger than 10. In smaller projects it might be more useful to make only an overview use case diagram but not all the use cases

Enterprise context: The use case approach is independent of the organizational environment

So this approach is selected in situations where user-level information is essential for the domain model, where variability should be expressed early and explicit and where overview information (as it can be found in use case diagrams) is needed. Of course the use cases do not solely form the domain model, other approaches like feature modeling or textual requirements should be chosen as additional modeling formalisms depending on the customization.

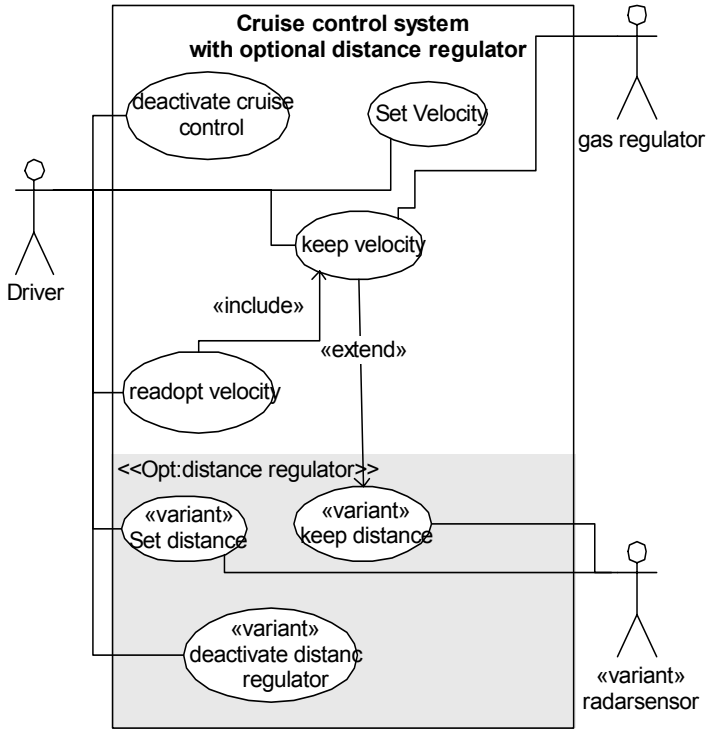
3.2 Variable Use Case Diagrams

In use case diagrams, any model element may potentially be variant in a product-line context. An actor is variant, for example, if a certain user class is not supported by a product. A use case is variant if it is not supported by some products in the family. A generic use case diagram for our cruise-control example is depicted by Figure 3. There, an optional distance regulator has been added, that is, three additional (and variant) use case have to be modeled by using the stereotype `<<variant>>`, as well as an additional actor, the optional radar sensor to measure the distance of a car in front. The additional feature has an impact on other use cases, which is modeled in the textual description of the affected use cases. An example for the use case "`keep velocity`" is given in the following subsection. Principally, variability should be written down on a level as low as possbile. If only parts of a use case are optional, not the whole use case should be marked as optional but only the parts in the textual description that really are optional. This helps in localizing variability for later phases. The places where variability occurs are then collected in the decision model

The new use cases that are included by adding a distance regulator can either extend existing use cases (as the "keep distance" use case extend the "keep velocity" use case) or can be handled as extra "standalone" use cases (like the "set distance" use case and the "deactivate distance regulator " use case). the third possibility is that optional use cases are included in existing use cases (no example here). The question under which circumstances which extension should be chosen is still unexplored and should be further investigated.

During application engineering, for each variant use case, it is decided whether the use case is (or is not) supported by the product to be built. The instantiation is done then with the help of the decision model. If a cruise control without distance regulator is built, all the variant use cases are removed, and the resulting use case diagram is the diagram shown in Figure 1.

Figure 3:Generic Use Case Diagram



3.3 Variable Textual Use Cases

In a textual use case description any text fragment may be variant. Variant text fragments are explicitly marked by pairs of the XML-like tags <variant> and </variant>. The decisions are integrated in the use case and underlined. Figure 4 shows an example of this approach, the use case "`keep velocity`"Decision modeling and instantiationWhether a use case in a use case diagram is an optional use case or whether it is an alternative to another use case is captured outside of the use-case diagram in a decision model. This is done simply because this information would overload the use-case diagram, make it less readable, and thus less useful.

The underlined questions in the use cases reflect the parts of the overall decision model that are relevant for this particular use case. This information is useful in both workproducts: integrated in the use case description, it helps to understand the use case's variability from the use case's point-of-view, in the decision model, it helps to understand the variability of the whole product family and what the impact of a particular variability is (e.g., it has an impact on this use case). Hence the overall decision model captures the relationship between the

decisions related to the above generic use case diagram and the textual description in Figure 4.

Figure 4: A Generic textual use case

Use Case Name: keep velocity
Short Description: keep the actual velocity value over gas regulator <variant> and control the distance to cars in front </variant>
Actors: driver, gas regulator, <variant> actor distance regulator </variant>
Trigger: actor driver, <variant> actor distance regulator </variant>
Precondition: --
Input: starting signal, velocity value v_{target}
Output: undefined
Postcondition: $v_{actual} = v_{target}$
Success guarantee: $v_{actual} = v_{target}$
Minimal guarantee: The car keeps driving
Main Sucess Scenario:

1.) <keep velocity> is selected by actor driver
2.) Does a distance regulator exist?
get v_{actual} (<Calculate Velocity>)
<variant OPT> get d_{actual} d_{target} (<Calculate Distance>) </variant>
3.) Does a distance regulator exist?
<variant ALT 1: no; only cruise control>
- compare v_{actual} and v_{target}
If $v_{actual} < v_{target}$: gas regulator increase velocity
- restart <keep velocity>
If $v_{actual} > v_{target}$: gas regulator decrease velocity
- restart <keep velocity>
else restart <keep velocity>
</variant>
<variant ALT 2: yes, cruise control + distance regulator>
- - compare vactual and vtarget
If $v_{actual} < v_{target}$ and $d_{actual} > d_{target}$: gas regulator decrease velocity
- restart <keep velocity>
else If $v_{actual} > v_{target}$and $d_{actual} < d_{target}$: gas regulator increase velocity
- restart <keep velocity>
else restart<keep velocity>
</variant>

That is, if the feature "`distance control`" is excluded the four variant use cases are removed and all variant text fragments are removed from the description of the use case "`keep velocity`", as well as the first alternative for step 3 is selected. This instantiation leads to the use case description given in the previous section.

3.4 Decision Modeling

Table 1 shows an excerpt of the decision model in textual form for the use case diagram and the textual use case. As in this example the variation point is rather simple (*Does a distance regulator exist or not?*) the decision model also is very simple. If there is more and more complicated variability and hierarchical decisions, the decision model gets more complex and can really support the software engineers during application engineering.

Table 1: Partial Decision Model

Variation Point	Decision	Actions
1	The car has no distance regulator	Remove Use Case "`set distance`" from use case diagram
		Remove Actor "`radar sensor`" from use case diagram.....
		remove variant <variant Opt > from use case "`keep velocity`" point 2
		remove variant <Alt 2> from use case "`keep velocity`" point 3.....
	The car has a distance regulator	Remove the <<variant>> tag from all use cases in the use case diagram
		remove the <variant Opt > tag and the </variant> tag from use case "`keep velocity`" point 2.....

4 Conclusions

In this report, we described how use cases can be applied for modeling the requirements for a system family. Therefore, we showed how a particular single-system use case approach can be extended to capture product line information and especially variability. The approach has been illustrated by the running example "cruise control system". In our experience, use cases are a good means to elicit, structure, and represent user-level information during the requirements phase. Extended with variation points, they also enable people to easily switch from single-system requirements-engineering practices to domain analysis. The produced generic use-cases also support and guide application engineering (in particular, its requirements phase). Thereby, each variation point must be instantiated, that is, each generic use-case with variation points is systematically transformed into a "normal" single-system use-case as it is expected by an individual customer. In general, the described approach pushes the explicit consideration of variability to the early phases of product-line development, which is required to systematically manage and evolve a product-line infrastructure. With the right decision model that documents the relationships and dependencies among variation points, the approach captures the traceability paths from variant use-case actions down to variant implementation elements.

Acknowledgements

The work described herein was based on work done by our Hiwis Jörg Dörr and Stefan Sollmann. Barbara Paech gave valuable comments on a preliminary version and helped improve this report.

5 References

[1] P. America and J. van Wijgerden. Requirements Modeling for Families of Complex Systems. In F. v. d. Linden, editor, *Third International Workshop on Software Architectures for Product Families*, LNCS 1951, Las Palmas de Gran Canaria, Spain, Mar. 2000. Springer.

[2] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wst, and J. Zettel. *Component-based Product Line Engineering with UML*. Component Software Series. Addison-Wesley, 2001.

[3] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. PuLSE: A Methodology to Develop Software Product Lines. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99)*, Los Angeles, CA, USA, May 1999. ACM.

[4] J. Bayer, D. Muthig, and T. Widen. Customizable Domain Analysis. In *Proceedings of the First International Symposium on Generative and Component-Based Software Engineering (GCSE '99)*, Erfurt, Germany, Sept. 1999.

[5] R. Biddle, J. Noble, and E. Tempero. Supporting Reusable Use Cases. In *Proceedings of the Seventh International Conference on Software Reuse*, Apr. 2002.

[6] A. Cockburn. *Writing Effective Use Cases*. Addison Wesley, 2001.

[7] H. Gomaa. Object Oriented Analysis and Modeling for Families of Systems with UML. In W. B. Frakes, editor, *Proceedings of the Sixth International Conference on Software Reuse*, June 2000.

[8] M. Griss, J. Favaro, and M. d'Alessandro. Integrating Feature Modeling with the RSEB. In *Proceedings of the Fifth International Conference on Software Reuse*, Vancouver, BC, Canada, June 1998.

[9] G. Halmans and K. Pohl. Considering Product Family Assets when Defining Customer Requirements. In K. Schmid and B. Geppert, editors, *Proceedings of the International Workshop on Product Line Engineering - The Early Steps: Planning, Modeling, and Managing*

(PLEES'01). Fraunhofer Institute for Experimental Software Engineering (IESE), Sept. 2001. IESE-Report No. 050.01/E.

[10] I. Jacobson. *Object-Oriented Software Engineering, A Use Case Driven Approach*. Addison Wesley, 1992.

[11] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse. Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.

[12] I. John and D. Muthig. Product Line Modeling with Generic Use Cases. In *Workshop on Techniques for Exploiting Commonality Through Variability Managemen, Second Software Product Line Conference, San Diego, USA, August 19-22 2002, http://trese.cs.utwente.nl/splc2-variability/*, Aug. 2002.

[13] I. John and D. Muthig. Tailoring Use Cases for Product Line Modeling. In *Proceedings REPL'02 International Workshop on Requirements Engineering for Product Lines,RE'02, Essen*, Sept. 2002.

[14] K. C. Kang, K. Lee, J. Lee, and S. Kim. Feature Oriented Product Line Software Engineering: Principles and Guidelines. In *Domain Oriented Systems Development – Practices and Perspectives*. Gordon Breach Science Publishers, 2002.

[15] Object Management Group. *OMG Unified Modeling Language Specification, Version 1.4*, September 2001.

[16] K. Schmid and T. Widen. Customizing the PuLSE Product Line Approach to the Demands of an Organization. In R. Conradi, editor, *Software Process Technology, 7th European Workshop, EWSPT'2000*, LNCS 1780, Kaprun, Austria, Feb. 2000. Springer.

Document Information

Title:	Modeling Variability with Use Cases
Date:	November 7,2002
Report:	IESE-063.02/E
Status:	Final
Distribution:	Public