Towards Integration of Software Case Reuse and Modelling of Variability

Katharina Wolter, Lothar Hotz, Thorsten Krebs
HITeC e.V. c/o University of Hamburg
Vogt-Kölln-Str. 30
22527 Hamburg, Germany
{hotz|krebs|kwolter}@informatik.uni-hamburg.de

Abstract

This paper introduces a novel approach to integrate reuse of software cases and dynamic variability modelling. Software cases comprise a complete definition of a software product and its development. This includes the problem description in form of a requirements specification as well as the solution in form of architecture, design and code. Previous software cases are identified based on the requirements specification and can be reused for different but similar problems.

The approach presented in this paper introduces a tight integration of reusing and modelling variability. During requirements specification, previously defined requirements can be reused and new requirements can be modelled – i.e. integrated into the variability model at the same time.

This paper describes work in progress. The ideas stem from research work within the EU-funded project ReDSeeDS (Requirements-driven Software Development)[1].

1. Introduction

The necessity to enhance reuse in software development is known. Software product lines and variability modelling are well-known approaches to reach this goal [1, 2, 7]. Adopting these approaches, the variability of the domain is modelled *in advance* to software development and reuse during application engineering.

In this paper, we propose an approach where reuse is based on software cases and variability is modelled *during* software development. Using this approach, the additional effort for variability modelling is kept minimal and only slight changes to ”traditional” software development processes are required. This approach can be seen as a complement or alternative to the strict separation of application and domain engineering in software product lines.[2]

According to [10] a *software case* comprises a problem statement (requirements) and a solution (design and implementation). The precisely defined requirements are mapped onto appropriate elements of the solution. Between the elements of the solution (architectural model, design model, code) a mapping is defined (see Figure 1). These mappings enable traceability from requirements to code elements. For a more detailed description of this approach we refer to [10].

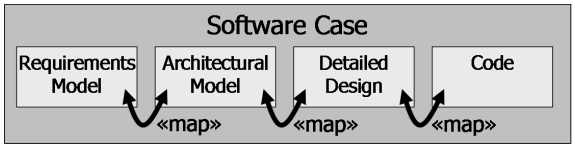

Figure 1. A Software case: Mapping requirements to architecture, design and implementation (taken from [10]).

All software cases are collected and summarized in one common *software knowledge model*. The single software cases do not contain variability. However, with each new software case additional variability is introduced in the software knowledge model. Software cases or parts of software cases can be reused based on their *similarity* to the software under development. Starting with some requirements, software cases with similar requirements can be retrieved and partially reused and / or adapted [10](see Figure 2).

In order to compute the similarity between software cases based on (partial) requirements specifications it is necessary to keep the software cases consistent. Each customer uses his own terminology, however. To keep requirements understandable for the customer, his terminology should be used in the requirements specification (e.g. scenarios). In order to ensure that the similarity between

[1] http://www.redseeds.eu

[2] For a definition of software product lines (SPLs) see e.g. [2]

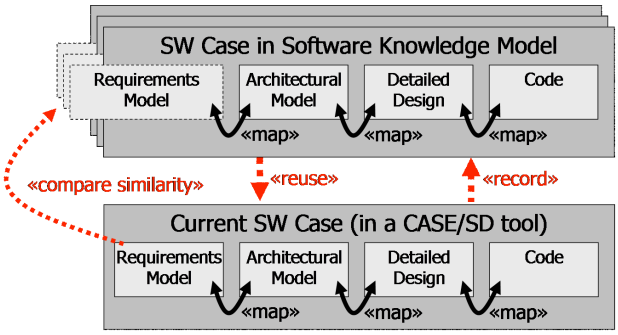

Figure 2. Reuse of software cases exploiting the variability defined in the software knowledge model (taken from [10]).

software cases can be computed these different terminologies can be managed in one *consolidated vocabulary*. During requirements specification this consolidated vocabulary is extended whenever a term is not yet defined or used with a different meaning.

Thus, in the consolidated vocabulary the variability of used terms and their relations to each other are summarized. Parts of the software knowledge model represent the consolidated vocabulary. Since the software knowledge model (including the consolidated vocabulary) can be very large, consistency ensuring tool support is essential for this task.

In the following we give some examples from the domain of a fitness club. This domain models the software to manage a fitness club, club members, staff members, classes and courses, the billing, etc.

The remainder of the paper is organised as follows. Section 2 illustrates how software development looks like using a software knowledge model defining the variability of all former software cases. Section 3 specifies the ingredients for this development process. Section 4 discusses the approach and compares it to other approaches, Section 5 summarises the main benefits.

2. The Vision

Using the ReDSeeDS approach [10], reuse already starts during requirements specification. New variability is introduced during software development, i.e. application engineering and domain engineering are closely integrated. In the following we illustrate how software development looks like using this approach.

The Requirements Engineer (RE) defines the requirements e.g. in a set of scenarios. Each scenario is a sequence of sentences written in restricted English. The sentences describing the system to be developed are called *requirements statements* or short *statements* in the following. A tool supports the RE during requirements specification. When the RE enters a new sentence the tool checks the software knowledge model for each term and displays the relevant part. The following cases need to be distinguished:

1. *The term is already defined in the software knowledge model*

   (a) *The term has the same meaning in the software knowledge model and the current software case*
   The term can be reused and the software knowledge model does not need to be changed.

   (b) *The term has a different meaning in the software knowledge model and the current software case*
   An example for this are *homonyms*. The term 'enter' for example can describe that a person goes into a room or that a person types in data. The software knowledge model needs to be extended with the new meaning.

2. *The term is not yet defined in the software knowledge model*

   (a) *The meaning is already defined in the software knowledge model using a different term*
   An example for this are *synonyms*. The term 'enter' for example is already defined but 'typing' is not. The software knowledge model needs to be extended with the new term.

   (b) *The meaning is not yet defined in the software knowledge model*
   The software knowledge model needs to be extended with the new term and its meaning.

The vocabulary used for specifying requirements of diverse customers can be very large. Thus, those vocabularies are consolidated and represented with the software knowledge model. By using this model, adaptation and maintenance of the vocabulary used in an organisation can be simplified.

The RE can use a query engine to search for similar software cases [10]. This search can be based on all requirements defined so far or on a subset that is of special interest. The RE can browse one or more of the most similar software cases and select requirements for reuse in the current software case. Because of the mapping between requirements, architectural models, detailed design models and code it is possible to reuse also the related parts of the problem solutions. This means that the RE can reuse parts of problem descriptions with the associated parts of the problem solutions from several former software cases. These partial problem descriptions with associated problem solutions are called *subcases* in the following.

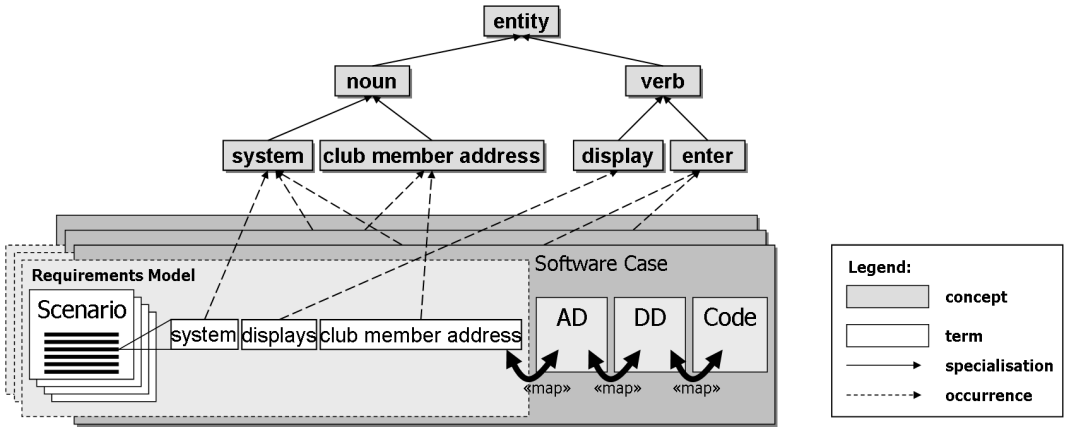

Figure 3. The software knowledge model.

In order to reuse subcases it is necessary to adapt them to the current software case and to integrate subcases from different former software cases. The mappings defined between the different models within one software case enable tool support for this task. As soon as the current software case development is finished, it is automatically included in the software knowledge model, i.e. no separate modelling activity is needed.

The software development process used in an organisation needs to be adapted to this approach. It is not necessary to establish a new development process, however. A main adaptation is in the activities a software developer has to perform. Besides traditional development, she has to compare new customer-specific terms to existing terms in the consolidated vocabulary and, if needed, extend this vocabulary, i.e. the software knowledge model, at appropriate places.

3. Ingredients

In this section we describe the software knowledge model containing all knowledge about software cases (see 3.1). Using the software knowledge model during requirement specification is sketched in 3.2. Section 3.3 describes how this model is reused and consistently evolved during software development.

3.1. Software Knowledge Model

The *software knowledge model* contains a representation for complete software cases: requirements, architecture, detailed design, and code. In this paper we present a first step towards this uniform representation of various information sources. We therefore limit ourselves to modelling the requirements in this paper.

However, we propose to use a generic logical representation for the software knowledge model and thus for representing variability (see [6, 11] for examples of such languages). In this model, *concepts* are used to represent requirements, components, source code files, and other artefacts that are used for software development (see Figure 3).

A *taxonomy* defines an inheritance structure between a concept and its specializations. Concepts can also be composed of other concepts. This is modelled in a *partonomy* that relates aggregates to their parts, respectively. Some concepts are also restricted by constraint definitions. These *constraints* can define exclusion (e.g. when concepts are incompatible to each other), inclusion (e.g. when the existence of a component is required), or other relations that constrain concepts and their properties.

These modelling facilities are used to realise the consolidated vocabulary. Figure 3 depicts how a taxonomy of terms used in requirement statements is modelled with concepts and specialisation relations. Furthermore, concepts are related to terms in the software cases they are used in, with the relation *occurrence* (i.e. *instance-of*).

3.2. Using the Software Knowledge model for Requirements Specifications

For clarifying how a consolidated vocabulary represented in a software knowledge model can be used for requirements specification, we extend the approaches described in [10] and [9].

A RE enters requirements for a software case e.g. in form of *scenarios*. A scenario contains requirements statements that are formulated in restricted English. In linguistic typology, *subject-verb-object (SVO)* (sometimes called *agent-verb-object (AVO)*) is a sentence structure where the subject (the agent) comes first, the verb second and the object third [5]. A statement in SVO looks like this: 'staff

member enters club member address'. There are also other approaches to restrict English text for expressing requirements, like the *Attempto Controlled English (ACE)*[3].

A *consolidated vocabulary* defines terms which can be reused to formulate requirements in form of statements. Structuring the vocabulary in a taxonomy has some key benefits:

- The vocabulary can be browsed to better identify reusable terms. The first distinction is made between nouns and verbs. A noun can be both, subject and object, depending on the role it plays in the given statement. Specializations for nouns are for example 'person', 'thing', etc. Specializations for verbs are for example 'moving', 'typing', etc.
- Homonyms can be distinguished by the specialization paths that lead to the terms. The two meanings for the term 'enters' for example can be identified by the path via 'moving' for 'club member enters fitness club' and 'typing' for 'staff member enters club member address'.
- The system's knowledge of a term is defined by the concept representing the term and it's relations (specialisation, parts and constraints).

3.3. Combining Evolution with Case-based Reuse

The software knowledge model representing the consolidated vocabulary enables better reuse of existing terms. The user interface for entering requirements statements can be enriched with a vocabulary browser such that the RE can browse the existing consolidated vocabulary while typing in a scenario.

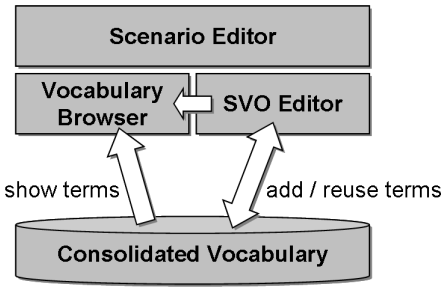

Figure 4. Scenario editor - consisting of SVO editor and vocabulary browser.

Figure 4 shows how a scenario editor can be realized, using the consolidated vocabulary. The editor consists of two main components:

- The *SVO editor* is used for entering requirement statements in subject-verb-object notation. Each of these statements belongs to a scenario.
- The *vocabulary browser* shows that part of the vocabulary, that is currently of interest. This means it filters nouns or verbs, or shows possible specializations of a term the RE has previously selected.

Entering statements, a RE can reuse terms from the consolidated vocabulary, or can enter new terms. The vocabulary browser shows the available terms and if the RE can not find a term describing what he wants to state, the editor can exploit the position of the browser to introduce the new term into the vocabulary at a specific position. For example, if the term 'staff member' is to be entered, and no related term can be found under 'person', the editor uses the browser position for "knowing" that a 'staff member' is a 'person'.

Another input method for new terms is to create relations to existing terms. Let us assume a RE wants to use the term 'employee', which is not modelled. He can browse the consolidated vocabulary and finds the term 'staff member', which is semantically the same. But both synonyms should be kept because 'employee' is part of the customer's vocabulary. Therefore, the term 'employee' can be entered into the consolidated vocabulary with a relation to 'staff member', which states that both terms are synonyms.

4. Discussion and Related Work

In case-based reasoning, *cases* (i.e. past experience) are stored to be retrieved when similar problems arise [8]. The processing required to analyze the experience is delayed until the time the case is retrieved for solving a new problem. At that time the problem is less haunting, because it is only needed to understand how the differences between the problem in the recalled experience and the current problem affect the solution proposed in the recalled experience. The solution proposed in that case is examined and applied to the current problem with suitable modifications.

Traditional case-based reasoning approaches also store the problem and the corresponding solution together in one case. But the information is not as well structured as it is in the ReDSeeDS approach. Here, a case contains requirements, architecture, detailed design and code – and most notably a mapping that represents transformations between these elements of a software case. Thus, the transformations are traceable; during development and after retrieving the case. This enhances identifying the corresponding (part of the) solution from a former cases because for every part of the solution (e.g. file, class, code fragment, etc.) it is known for which requirement it has been developed.

Software Product Lines (SPLs) are a means for large-scale reuse. A product line contains a set of products that share a common, managed set of features satisfying the specific needs of a particular market segment or mission [1, 2]. Development effort is distributed over customers by managing an asset store and reusing assets for different customers.

In SPLs, domain engineering plays a key role. This includes analysing the product domain, building a common architecture and planning reuse – in the sense that combinations of reusable assets are managed. Domain engineering is done a-priori to product development. The approach described in this paper enables dynamic extension of the reusable asset repository. By integrating reuse and evolution, reuse does not have to be planned a-priori. Evolving the model during product development contains both, evolving the problem description (vocabulary, scenarios, etc.) and including the solution to the problem (i.e. architecture, design and code).

SAMOVAR (Systems Analysis of Modelling and Validation of Renault Automobiles) aims at preserving and exploring the memory of past projects in automobile design [4]. A so-called Problem Management System (PMS) contains structured knowledge about problem definitions and with this enables the user to search and find similar problem descriptions that have been solved in the past. The system relies on building ontologies, semantic annotations of problem descriptions relatively to these ontologies, and the formalisation of the ontologies and annotations.

The SAMOWAR approach is similar to the one described in this paper in the sense that one model of past cases is built and this model is used to query and find past cases with similar problem descriptions. However, this approach does not include dynamic extension of this model during product development. This integrated reuse and evolution is one of the key benefits of our approach.

The AMPL (Asset Modelling for Product Lines) language developed in the ConIPF (Configuration in Industrial Product Families)[3] project defines a language to fully represent product structures (incl. features, context, software and hardware artefacts) [6]. These modelling facilities can be extended to also formalize requirements definitions (e.g. use cases, scenarios, statements) and architecture, design and code.

5. Summary

In this paper, we have introduced a novel approach that integrates reuse of software cases and variability modelling. A software case comprises the problem description in form of a requirements specification and a solution description in form of architecture, detailed design and code artefacts. Reusing software cases is enabled with a query engine that compares the requirements specification of past cases and the current problem. Variability is modelled implicitly by the set of past software cases.

Requirements are specified through scenarios that consist of requirement statements formulated in restricted English; namely subject-verb-object (SVO) [9]. Our approach explicitly supports the definition of statements in SVO by managing a consolidated vocabulary. This vocabulary is represented with a logic-based software knowledge model, where each term is represented by a concept. While defining new statements, the concepts in this model (which are defined through former cases) can be reused and new concepts can be entered dynamically. This integration of requirements specification and evolution of the modelled reusable variability leads to dynamic domain engineering. Our approach therefore complements the static (a priori) variability modelling that is typical for product development in software product lines.

The approach described in this paper is close to ”traditional” software development processes. Dynamic domain engineering has the benefit that evolution of the product line does not have to be planned before product development takes place. Evolving the reusable artefacts can be done during product development.

Traditional case-based reasoning is concerned with identifying one case or more cases from the case library that are most similar to the current problem. Afterwards these cases are modified according the new requirements. Using our approach it is possible to reuse subcases – i.e. parts of stored software cases. Subcases from different former cases can be identified and combined for reuse.

Acknowledgement This work is partially funded by the EU: Requirements-driven Software Development System (ReDSeeDS) (contract no. IST-2006-33596 under 6FP).

The project is coordinated by Infovide, Poland with technical lead of Wasaw University of Technology and with University of Koblenz-Landau, Vienna University of Technology, Fraunhofer IESE, University of Latvia, HITeC e.V. c/o University of Hamburg, Heriot-Watt University, PRO DV, Cybersoft and Algoritmu Sistemos.

References

[1] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, and K. Pohl. Variability Issues in Software Product Lines. In *Proc. of the Fourth International Workshop on Product Family Engineering(PFE-4)*, Bilbao, Spain, October 3-5 2001.

[2] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[3] N. Fuchs, U. Schwertel, and R. Schwitter. Attempto Controlled English (ACE). *Language Manual, Version 2.0*, 1992.

[3] http://www.conipf.org

[4] J. Golebiowska, R. Dieng-Kuntz, O. Corby, and D. Mousseau. Building and Exploiting Ontologies for an Automobile Project Memory. In *Proc. of First International Conference on Knowledge Capture (K-CAP)*, Victoria, BC, Canada, October 23-24. 2001. ACM.

[5] I. Graham. Task scripts, use cases and scenarios in object oriented analysis. *Object Oriented Systems*, 3:123–142, 1996.

[6] L. Hotz, K. Wolter, T. Krebs, S. Deelstra, M. Sinnema, J. Nijhuis, and J. MacGregor. *Configuration in Industrial Product Families - The ConIPF Methodology*. IOS Press, Berlin, 2006.

[7] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented Domain Analysis (FODA) Feasibility Study. *Technical Report CMU/SEI-90-TR-021*, 1990.

[8] M. Sasikumar. Case-based Reasoning for Software Reuse. In *Knowledge Based Computer Systems-Research and Applications (International Conference on Knowledge-Based Computer Systems)*, pages 31–42, Bombai, India, December 12-15 1996. Narosa Publishing House, London.

[9] M. Smialek. Acoomodating Informality with Necessary Precision in Use Case Senarios. *Journal of Object Technology*, 4(8):59–67, 2005.

[10] M. Smialek. Towards a Requirements driven Software Development System. In *Models 2006*, 2006.

[11] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen. Towards a General Ontology of Configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (1998), 12*, pages 357–372, 1998.