Can precise requirements models drive software case reuse?

Albert Ambroziewicz, Jacek Bojarski,
Wiktor Nowakowski, and Tomasz Straszak

Warsaw University of Technology,
Warsaw, Poland
{ambrozia,bojarsj1,nowakoww,straszat}@iem.pw.edu.pl
http://www.redseeds.eu/

Abstract. A crucial issue for approaches towards software reuse is the formulation of artefacts to be reused. It is necessary to introduce appropriate formalisms for organising artefact sets and artefact structure in order for the reuse process to become effective. This includes also the possibility to reuse partial artefacts thus enabling their easier adaptation to the current problem and merging with the newly built system. This paper introduces the notion of software cases which define coarse structures for software artefacts linked through MDA style mappings. Software cases can be sliced thus offering partial artefacts with fine grained structure of individual models expressed through appropriate languages. The presented approach is highly requirements driven – all the slices originate in requirements artefacts which are traced to other layers: architecture, design and code.

1 Introduction

A fundamental issue in any engineering approach towards reuse is identification of reusable assets (artefacts). Reusable elements should somehow allow for repetition of their substance or structure. In software engineering, reuse efforts originate in ideas taken from circuit technology (integrated circuits - see [10]). Unfortunately, identification of repeatable "software ICs" is very difficult and usually fails, due to variability and broadness of problem domains (see e.g. [13]). Despite this, the above analogy holds when considering interoperability of artefacts (as for the interoperability of ICs). Two issues emerge here: manifestation of artefacts and their compatibility. In order for artefacts to be reused they need to be manifested in a way enabling comparison. For the artefacts selected for reuse, their technical compatibility with the currently built system is mandatory.

Certain level of interoperability between software assets is introduced in meta-modelling approaches where the artefacts (models or code) are based on a common schema or modelling language. Unfortunately, meta-modelling or model-driven approaches usually lack the manifestation element or manifestation artefacts have to be built separately adding significantly to the overall effort. Despite this, it can be noticed that models and code produced during a typical

software project form a complete artefact that has the potential to be reused. A major problem here is to be able to identify such a legacy artefact, and then to identify its "parts" that are suitable for the currently solved problem. This should be possible no matter which level of abstraction (whole system, component, procedure) for the legacy artefact is suitable to be used again.

It can be noted that all individual artefacts resulting from a software project form a specific "case" that combines a specification of a particular problem (user requirements) together with its solution (design and code). We can call it a "software case". Any specific software case is in fact "manifested" through its specific requirements. Requirements (mainly - functional ones) can also serve for dividing the whole case into certain "sub-cases" or "slices". These slices can be extracted from the whole thus enabling partial reuse. It has to be stressed that this "slicing" of software cases is possible only when certain mechanisms for mapping between artefact layers are in place. This makes it necessary to define precisely the structure of software cases which includes the structure of individual elements and mappings (realised through model transformations) between them. With such precisely defined structure, individual ICs (software components, code pieces) can be identified for reuse through comparing requirements and tracing into these ICs. Moreover, whole frameworks of software components forming partial solutions can be extracted and reused.

Concepts based on identifying parts of artefacts to be reused are wide-spread and common in software engineering. Also, approaches where requirements are used as compositional elements during reuse have been already researched. All these approaches seek ways to achieve efficient reuse, although certain problems identified already by Basili [1] still hold. Because of the size of this paper, for detailed discussion of reuse methods we refer to [2] [7] where many approaches are compared and current state-of-the-art is discussed. This work refers to case-based approaches presented e.g. in [6], [9]. Isolation of sub-cases from complete cases is presented in [8].

In this paper we describe the above idea in detail. The essence and ways to construct software cases is described in Section 2, while the concept of slices is explained in Section 3. Mechanisms for applying software cases in reality are presented in Section 4. Paper concludes by presenting the results of evaluation performed so far, based on prototype tools, and summarising ideas for the direction of future research (Section 5).

2 Software cases – coarse-grained elements for software reuse

The previous section mentions a variety of approaches to case-based reuse. In the context of software engineering, we can define software-case reuse. What we in fact want to reuse are sets of logically, functionally and structurally related artefacts forming a solution to a problem we currently have at hand. Thus, a software case (SC) – can be defined as a container for a solution to a precisely expressed problem in the form of a requirements model. All elements of the

problem statement are mapped onto appropriate elements of the problem solution thus making the solution reusable through problem matching. The reusable elements of the solution are design models (including architectural model and detailed design model), and the final code. This is illustrated in Figure 1, where relationships between four major elements of a software case and their more detailed structure are presented.

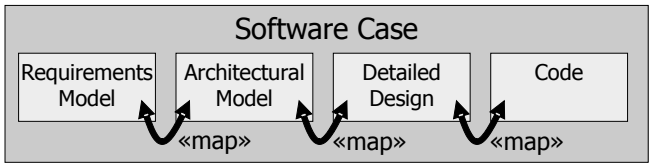

Fig. 1. Relationships between requirements, architecture, detailed design and code in a software case

To describe software cases we should use a standard language which we shall call the Software Case Language (SCL) – see Figure 2. SCL itself decomposes into a set of languages describing individual "layers" of a software case – the Requirements Specification Language (RSL, formally defined in [3]), the Software Development Specification Language (SDSL, defined in [4]) and of course a selected programming language.

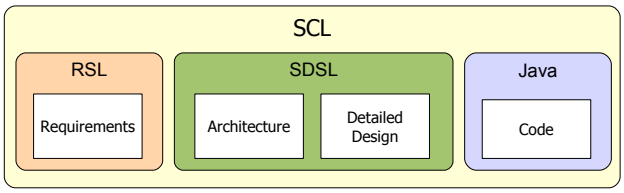

Fig. 2. Languages composing the Software Case Language

RSL is a language which introduces several new concepts, however most basic constructs (like use cases) are adapted from UML. In RSL a strict distinction between requirements-as-such and their representations is made. Requirements representations have a rigourous meta-model, which allows meaningful comparison of requirements contents and can drive reuse by determining parts of complementary requirements specifications (see [14]). Each of the requirement (use case) representations carries the same structured information.

RSL gives also means to define and visualise a vocabulary of the system domain. Such a vocabulary contains all the business domain information while requirements representations describe pure functionality of the system and constraints, with no interleaving of domain elements definitions. Requirements specification written in RSL separates user – system interactions from definitions of concepts in the business environment and allows for coherency of its content by linking requirements' sentences to precisely defined nouns and phrases. The

domain vocabulary maps contained elements to global morphological dictionary, making it a base for semantic comparison between texts written in RSL.

The second element of SCL – the SDSL is not a completely new language. The Unified Modelling Language (UML) [11] [12] is currently the most popular and widely used language for modelling architecture and design. It was thus reasonable to introduce SDSL as a subset (profile) of UML together with some guidelines for using it. The primary concern while selecting elements from UML to be used in SDSL was to make it sufficient to express typical architecture issues and at the same time simple enough to keep a coherency with RSL and transformation from requirements to other layers[1]. To express elements of the static part of an architectural model, a component diagram is used. Suitable UML elements for a component diagram are: component, interface, dependency, class, and package. Dynamics on the level of architecture are presented in the form of sequence diagrams. Suitable elements for sequence diagrams are: lifelines and messages.

Detailed design of a software system is the lowest "abstraction level" of the project specification. It should contain all the logical elements for every component in the relevant architectural specification. Detailed design model is the basis of implementation in a concrete programming language (e.g. Java or C#). Such models are expressed in UML through class diagrams. Similarly to the architectural level, we can represent dynamics of sub-systems through sequence diagrams. Suitable UML elements for a class diagram are: class, interface, data type, association, aggregation, composition, generalisation, realisation. Suitable elements for a sequence diagram are: lifeline, message, and combined fragment.

All artefacts described using SCL can be connected through traces, thus showing dependencies among them. Inter-layer traces give information about the existence and nature of the elements implied by elements of the previous (higher) layer. For instance, architectural-level sequence diagrams trace from requirements-level use cases, or detailed design classes trace from components they implement. Intra-layer traces are dependencies between elements of the given layer inter-related with each other. For example, vocabulary elements relate to specific requirement representations where these elements (terms) are used.

Having such a coherently inter-linked structure of software cases we can try to divide them into smaller fragments. These smaller fragments – subcases or slices – can serve for determining parts of a software case for more fine-grained reuse. Since all the constructed software cases were created using the same language (SCL), they have the potential for being merged into the current software case also specified using SCL. This is described in the following sections.

3 Slicing – creation of sub-cases

The idea of a "slice", as a part of a program, appears in [15]. If we adapt this definition in the context of requirements-driven and case-based reuse, this could be rephrased to:

[1] Transformations are not covered in this paper.

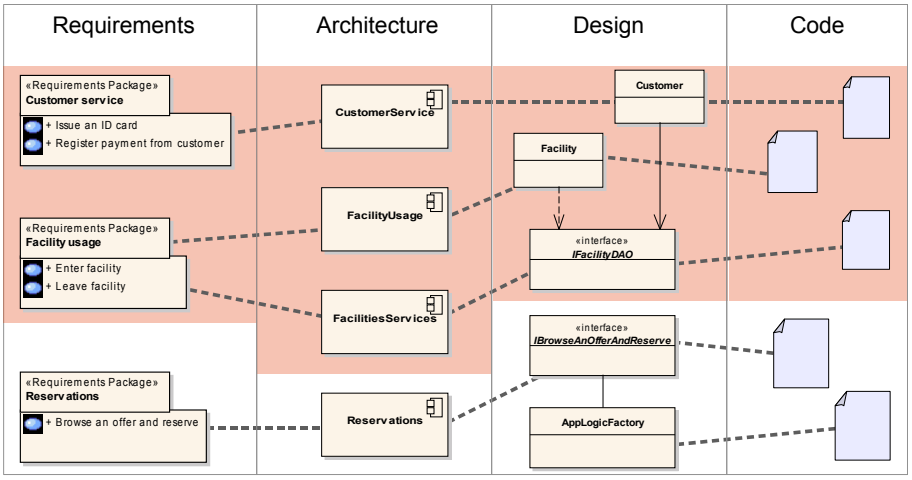

Fig. 3. Software case slicing

Software Case slicing is a decomposition technique that extracts from Software Case, artefacts relevant to a particular set of requirements. Informally, a slice provides the answer to the question "What artefacts realise the system behaviour described by given set of functional requirements?"

Main analogy between program slicing and SC slicing is that the basis for a slice is either a set of computations or requirements. By selecting a SC slice one actually selects another, smaller SC – a sub-case of the original (source) SC (see Figure 3). This sub-case is a complete SC itself, as it conforms to the definition from the previous section. It is important to define a set of rules used for determining the scope of a slice within the source SC. The "basic" (minimal) slice would be composed of selected requirements, their nested elements and intra- and inter-layer dependencies of selected requirements. This minimal software cases would not be sufficient for reuse in some situations however, as elements used by elements present in such a slice would not be selected, having the whole solution incomplete (e.g. missing dictionary elements not connected to selected requirements, but used in definitions of notions present in a slice). Of course, "exploding" a slice by incorporating in it any element related in any degree to already used element (creating a "maximal" slice) is not a solution too, as in many cases it would lead to sub-cases nearly identical to the source SC (especially when spaghetti-like or incorrectly decomposed systems are concerned).

One of the possible solutions seems to be to use a tool presenting to the user an automatically computed minimal and maximal slice and a proposition of near-ideal slice, where dependencies among elements are broken or left untouched according to the set of rules. "Hard" dependencies – the "unbreakable" ones (all traceability inter-layer links having source in slice-forming requirements) would be then added to all potential slice elements which could be attached

manually to a slice (marked by lower degree children of core slice elements) – "soft" dependencies. This way the user would build a reusable asset from the minimal slice, or by removing optional dependencies, rather reduce the maximal slice to the "ideal" one.

The most basic algorithm for calculating slice offers the following taxonomy of dependencies: "hard dependencies" are all traceability links directly connecting selected requirements with architecture model elements, these architecture model elements with their parent elements and implementations in detailed design model elements, these detailed design classes with code artifacts; also, on the level of requirements: links between phrases used in scenario sentences and notions from domain vocabulary. All other traceability links are considered "breakable".

The role of a user maybe essential in forming of pre-calculated slice, as this process is based mostly on functional requirements or operationalised non-functional requirements (in both situations: use cases). For this to be achieved, a system of intuitive and convenient slice/SC visualisation should be implemented (a proposition of this kind of user interface is shown in Figure 4).

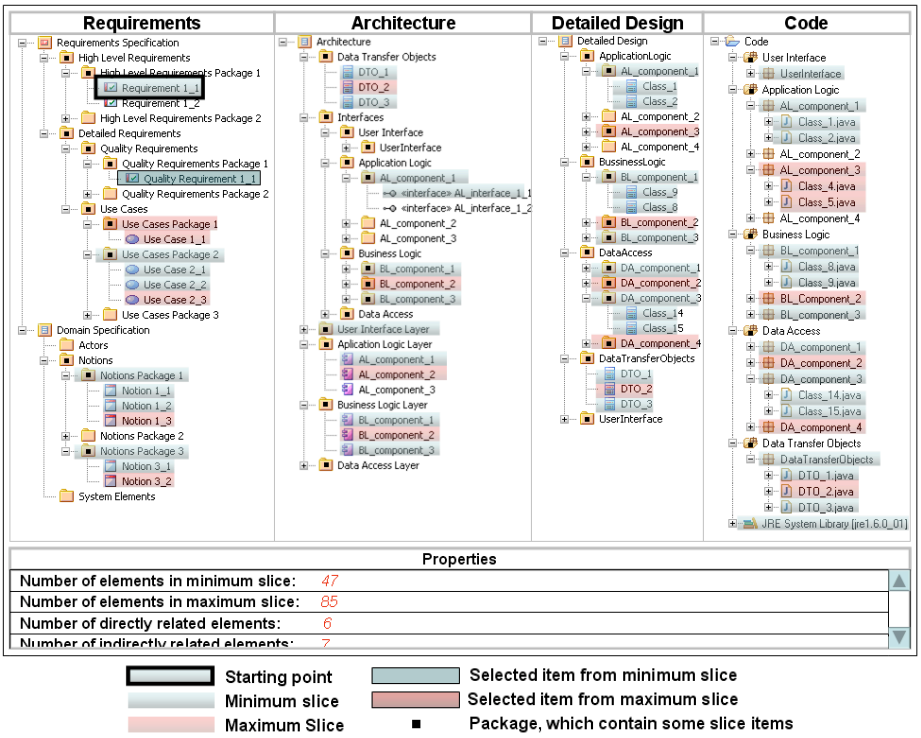

Fig. 4. Slice Visualisation

More detailed proposition of a system implementing the presented ideas is described in the following section.

4 Engine for reuse

To enable case-based reuse as described theoretically above, comparable and sliceable software cases should be stored in a SC repository. This repository would have an interface to a query engine that allows for searching for solutions fulfilling the search criteria which are simply formed of sets of requirements.

The whole case-based reuse process can begin when an analysts sketches a draft requirements specification (accompanied by a domain vocabulary), forming an input to the query engine. Then, a similarity between the analyst's requirements model and the repository contents is calculated, and list of search results is presented to the user. The analyst can choose SC/slices matching the formulated search criteria and incorporate them into the currently developed system.

The same mechanism can be used to compare successive versions of the same SC allowing for effective change management. Change tracking can be used to compare versions of automatically generated and edited layers of a SC. In this approach certain models created in the first iteration are then reused for development in the second iteration. It has to be noted that only relevant slices of the first-iteration software case can be used. This is controlled by changes made to the requirements model in the second iteration.

5 Conclusions and future work

This paper presents methods for requirements based slicing of software cases and for visualizing slices what significantly support software case reuse. We can label the sketched approach as "requirements-driven software case reuse". Software cases are manifested (exposed) through their requirements. Requirements are the elements compared for reuse. The artefact that is then reused is a relevant slice of a software case most similar to the currently built one. It is important that the reusable asset repository introduced in this paper is based on a uniform specification language. This allows for formulating precise requirements models traceable into design and code artefacts.

The ideas presented in this paper are being currently researched and validated in the scope of the ReDSeeDS project (see Acknowledgments). The presented language for building software cases has already been validated. Industry Partners within ReDseeDS have used this language to build real life examples by their analysts and developers. Validation results have been published in [5]. So far, a standard UML tool (Enterprise Architect from SparxSystems) has been used to create appropriate artefacts. This is a satisfactory solution for SDSL. Currently, a dedicated tool for RSL is tested to substitute the UML tool profile.

Future work includes building a comprehensive query engine where the stored software cases could be compared and marked for differences (with appropriate slices visualised). Retrieval mechanisms most suitable and effective for software cases defined through SCL are currently investigated. Results of this research shall be implemented in the currently built engine.

Finally, it can be noted that the presented approach, apart from contributing to the reuse field, is a solution allowing for analysis of modularity of software

systems, supporting change management (enabling analysis of how do changes in the requirements model influence changes in other software case layers) and traceability links on the path from requirements to code.

Acknowledgments. 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 Warsaw 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. Victor Basili and H Dieter Rombach. Support for comprehensive reuse. *IEEE Software Engineering Journal*, 6(5):303–316, 1991.
2. E S de Almeida, A Alvaro, D Lucredio, V C Garcia, and S R de Lemos Meira. A survey on software reuse processes. In *IEEE International Conference on Information Reuse and Integration*, pages 66–71, 2005.
3. Kaindl et al. Requirements specification language definition. Project Deliverable D2.4.1, ReDSeeDS Project, 2007. www.redseeds.eu.
4. Kalnins et al. Reuse-oriented modelling and transformation language definition. Project Deliverable D3.2.1, ReDSeeDS Project, 2007. www.redseeds.eu.
5. Krebs et al. Modelling and transformation language validation report. Project Deliverable D3.4, ReDSeeDS Project, 2007. www.redseeds.eu.
6. Gilles Fouqué and Stan Matwin. A case-based approach to software reuse. *Journal of Intelligent Information Systems*, 2(2):165–197, 1993.
7. William B. Frakes and Kyo Kang. Software reuse research: status and future. *IEEE Transactions on Software Engineering*, 31(7):529–536, July 2005.
8. Andreas Jedlitschka and Markus Nick. Scenarios, representation, and usage issues for software case-oriented comprehensive reuse. In *Proceedings of the International Workshop on Model Reuse Strategies (MoRSe 2006)*, pages 1–4, 2006.
9. Panagiotis Katalagarianos and Yannis Vassiliou. On the reuse of software: A case-based approach employing a repository. *Automated Software Engineering*, 2(1):55–86, 1995.
10. M D McIlroy. Mass produced software components. In P Naur, B Randell, and J N Buxton, editors, *Software engineering concepts and techniques, Proceedings of NATO Conference on Software Engineering*, pages 88–98, New York, 1969.
11. Object Management Group. *Unified Modeling Language: Infrastructure, version 2.0, formal/05-07-05*, 2005.
12. Object Management Group. *Unified Modeling Language: Superstructure, version 2.0, formal/05-07-04*, 2005.
13. Douglas C Schmidt. Why software reuse has failed and how to make it work for you. *C++ Report*, 11(1), January 1999.
14. Michał Śmiałek. Can use cases drive software factories? In *Workshop on Use Cases in Model-Driven Software Engineering (WUsCaM)*, 2005.
15. Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, 1984.