Mechanisms for requirements-based model reuse

Michał Śmiałek

Warsaw University of Technology, Warsaw, Poland
smialek@iem.pw.edu.pl

Abstract. The paper presents a sketch of a methodology for model-reuse- oriented software development. It describes a methodological core, reuse enabling interface and reuse methodological plug-in. These elements form a methodological basis for the construction of an automatic software case modeling, mapping and retrieval tool.

1 Introduction

Contemporary software systems become more and more complex. The complexity of these systems is associated with the complexity and changeability of problems specified through requirements specifications. Complex problems lead to even more complex solutions implemented in the technological space that changes even faster than the problem domain. Despite this complexity (or maybe - due to it), vast majority of software development projects seem to ignore past knowledge about solving specific problems. This might be explained by significant difficulties to reuse knowledge in such a complex domain as software engineering is. The main obstacle here is that there is no standard way to capture knowledge about complete cases leading from the problem (requirements) to its solution (design and code). There also seem to be no effective mechanisms to find and reuse past solutions to problems similar to the currently solved one [1]. In this paper we will present a sketch of comprehensive methodology (see [2] for an initial insight, and [3] for a current survey) enabling comprehensive reuse mechanisms in a software development organization.

2 Vision for organizing software reuse

It can be argued that the main problem with existing software reuse strategies is the size of investment that needs to be undertaken to enable the reuse process (see [4]). Thus, while designing a new reuse strategy we need to consider several topics important from the point of view of software development teams.

- **Compatibility with the existing software development processes.** The new strategy should easily fit into the current practice of software organizations. It should add as little additional activities as possible.
- **Additional effort needed when formulating software knowledge.** Ideally, the process of formulating software knowledge should be seamless from the point of view of software developers. This means performing exactly the same set of activities and using the same languages and tools as without the reuse strategy in place.

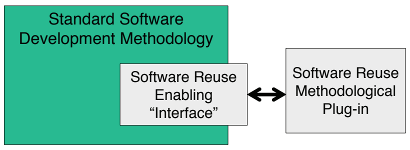
Standard Software Development Methodology
Software Reuse Enabling "Interface"
Software Reuse Methodological Plug-in

Fig. 1. Two core elements of a reuse-oriented software development methodology

- **Ease of capturing software knowledge for reuse.** It could ideally mean literally copying the current project workspace to an appropriate library. The cost should be minimal enough to make saving current software cases in a reuse library - an obvious routine.
- **Ease in reusing past software knowledge.** We should postulate that as for capturing software cases, the efforts associated with reusing them should be kept to the minimum. This means that querying a library of past cases should be based on the artifacts that would be developed anyway in a normal development practice. Moreover, query results should give precise pointers to these places that necessitate rework thus minimizing efforts associated with tailoring.

We need a reuse system that is compatible with the state-of-the-art software development methodologies, and significantly reduces efforts in formulating, capturing and reusing software knowledge. This system should ensure that the initial investment in changing practices will be as small as possible. Moreover, in case the organization invests in implementing a new general software development methodology, the investment in additionally setting a reuse strategy should be dismissably small. Also the additional investment in formulating reusable software knowledge should be small enough to be practically dissolved in other activities.

3 Adding reuse activities to a modern software development lifecycle

Existing general purpose software development methodologies like formal RUP [5] or agile XP [6] and FDD [7] do not offer guidance for applying software reuse. On the other hand, current trend in reuse oriented methodologies assumes special product-line oriented software development lifecycle (see eg. [8]). In order to fulfill the above vision we need a process that is compatible with general methodologies widely used in various software development companies. It can be noted that these methodologies are usually requirements-driven (use cases in RUP, user stories in XP, features in FDD). Thus, the basis for reuse should lie in finding similar requirements and reusing related design models and code.

Having this important commonality between methodologies we can try to identify the area where these methodologies can be extended with a reuse strategy. From this point of view, the base software development methodology has to enable maintaining "organizational memory" based on software requirements. This means that the artifacts created during development, starting from requirements, have to be captured in electronic tools. Moreover, the structure of these artifacts has to be repeatable between projects in order to make their reuse feasible. It can be noted that capturing artifacts in

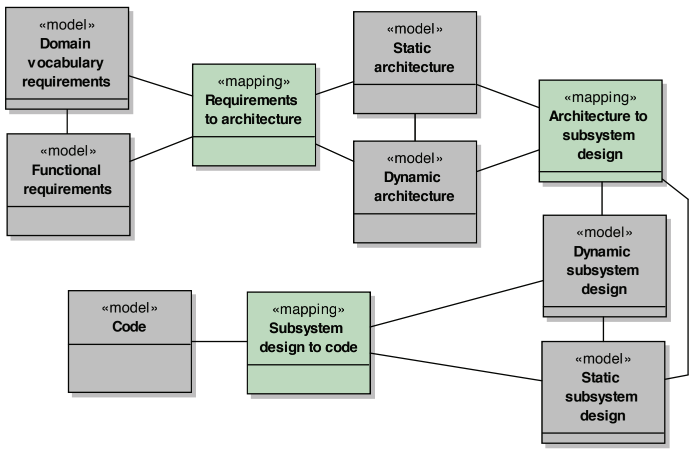
«model» Domain vocabulary requirements
«model» Functional requirements
«mapping» Requirements to architecture
«model» Static architecture
«model» Dynamic architecture
«mapping» Architecture to subsystem design
«model» Dynamic subsystem design
«model» Static subsystem design
«model» Code
«mapping» Subsystem design to code

Fig. 2. Metamodel showing the structure of high level elements of a software case

a standard way does not yet constitute a reuse strategy. It can be treated as an enabler of such strategy, or a "software reuse enabling interface" to those activities that form the reuse process. This is illustrated in Figure 1. Thus, when describing our reuse-oriented lifecycle process we will clearly distinguish activities forming the reuse enabling interface and the reuse plug-in.

4 Reuse Enabling Interface and Methodological Plug-in

The Reuse Enabling Interface basically consists of a software case specification language. This language would allow to specify cases produced during normal development lifecycle as briefly described in Section 3. In Figure 2 we can see the main pieces forming a software case, i.e. the elements that constitute technical knowledge collected during a software development project and possibly reused in future projects. It can be noted that metaclasses representing highest level elements of a software case are stereotyped as «model»s and «mapping»s. The first of these stereotypes distinguishes these pieces of knowledge that form complex abstractions of a certain domain or software system. These abstractions are formed of several interconnected elements (model elements) and allow for defining the system in a way understandable to the readers and possibly allowing for automatic processing. It can be noted that code is treated here also as a kind of model (written on the lowest level of abstraction). The second stereotype is attached to these software case elements that form descriptions of ways to translate (change) one «model» into another «model». This includes specifications to translate these models automatically or manually created links between model elements.

Having the above reuse enabling interface language we can define activities that shall allow for reuse of software cases specified in this language. Below we present most important of such activities that can be plugged into a normal software development process forming the Reuse Methodological Plug-in.

– **Find a software case** - query for software cases in a repository by finding similarities between cases based on similarity between their functional requirements and domain vocabulary requirements.

– **Merge a software case into current workspace** - retrieve various «models» associated through «mappings» with found similar requirements.
– **Show differences between software cases** - mark «models» mapped from similar requirements with differences to be handled by developers.
– **Store a software case** - copy the current software case into a software case repository.

It has to be stressed that the above activities should be implemented so that minimal possible effort is needed to realize them, especially when we consider iterative and agile lifecycle process as mentioned in Section 3. This necessitates the use of an extensive automatic tool that implements appropriate model mapping and retrieval mechanisms (see [4]).

5 Conclusions

The idea presented in this paper can be the basis for constructing a complete Requirements Based Software Development System (ReDSeeDS for short). This system should allow for realizing the vision described at the beginning through presented reuse mechanisms. The requirements for the developed system should certainly be derived from the briefly described lifecycle process consisting of base methodology, reuse enabling interface and reuse plug-in.

References

1. Schmidt, D.C.: Why software reuse has failed and how to make it work for you. C++ Report **11** (1999) – http://www.cs.wustl.edu/ schmidt/reuse-lessons.html.
2. Basili, V., Rombach, H.D.: Support for comprehensive reuse. IEEE Software Engineering Journal **6** (1991) 303–316
3. de Almeida, E.S., Alvaro, A., Lucredio, D., Garcia, V.C., de Lemos Meira, S.R.: A survey on software reuse processes. In: IEEE International Conference on Information Reuse and Integration. (2005) 66–71
4. Llorens, J., Fuentes, J.M., Prieto-Diaz, R., Astudillo, H.: Incremental software reuse. Lecture Notes in Computer Science **4039** (2006) 386–389
5. Kruchten, P.: The Rational Unified Process: An Introduction, 3rd ed. Addison Wesley (2003)
6. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (2000)
7. Palmer, S.R., Felsing, J.M.: A Practical Guide to Feature-Driven Development. Prentice Hall PTR (2002)
8. Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D Paech, B., Wüst, J., Zettel, J.: Component-based Product Line Engineering with UML. Addison-Wesley (2001)