Can Use Cases Drive Software Factories?

Michał Śmiałek

Warsaw University of Technology and Infovide S.A., Warsaw, Poland
smialek@iem.pw.edu.pl

Abstract. Contemporary software systems are becoming more and more complex with many new features reflecting the growing user's needs. Software development organizations struggle with problems associated with this complexity, often caused by inability to cope with constantly changing user requirements. Supporting efforts to overcome these problems, this paper proposes a method for organizing the software lifecycle around precisely defined requirements models based on use cases that can be quickly transformed into design level artifacts. The same use cases with precisely linked vocabulary notions are the means to control reuse of artifacts, promising the lifecycle to become even faster and more resourceful. With properly applied use case models we can significantly improve the concept of "software factories" that newly emerges in the area of model driven software engineering.

1 Introduction

In the IT industry it is a well known fact that software development organizations struggle (generally without success) to produce software systems consistent with constantly changing needs of their clients. While changing, together with changes in the businesses, these needs are usually communicated in a very ambiguous and imprecise way. Such poor quality requirements are then translated by architects, designers and programmers into a software system. Key issues here are time from requirements to the resulting system, quality of the system[1] and capability of coping with changes in user's needs during the current project and after it has finished. Unfortunately, these issues raise many problems associated with inability to cope with change and complexity of the developed systems.

Humans deal with complexity by using abstraction. Most prominent way of applying abstraction in software engineering is creating models of requirements and models of the system (architecture, code). Complex subsystems (components) can be encapsulated and used by other subsystems through much simpler interfaces - their complexity is thus abstracted away. Abstraction can be also applied by using patterns that solve commonly occurring problems.

These ideas for managing complexity of software systems were formed into a general idea of software factories [1]. In such a "factory" more general (abstract) models are transformed into more specific ones, and this "product line"

[1] By quality we mean simply the level of client's satisfaction through meeting his/her needs.

of transformations leads us finally to executable artifacts (not necessarily textual code, perhaps - Executable UML [2]). The product lines rely heavily on reuse of artifacts like frameworks, patterns or individually encapsulated executable components.

Software factories seem to be a promising newly emerging software development methodology that partially builds on the ideas of MDA [3]. It has to be noticed that the methodology is consistent with and tries to take valuable elements from both formal methodologies like UP [4] and agile methodologies [5] like XP [6]. However, there seem to be two crucial problems not tackled properly by software factories. The first problem is associated with the fact that the software factory method concentrates mostly on dealing with design and implementation. It leaves unresolved the problem of keeping requirements synchronized with the remaining artifacts, thus leaving the "requirements gap" [7, 8] still open. The second problem is with lack of guidance on applying reuse efficiently. It is silently assumed that the software developers have the knowledge and resources to hand-pick appropriate reusable assets.

In the paper we will propose a method for dealing with the above problems by applying a variant of use case driven development lifecycle [9]. The presented method is intended to close the gap between the user's needs and the resulting system by providing a standard notation for use case content and a transformation method to enable fast and resourceful lifecycle for system development based on reuse of artifacts (see [10] for additional insight). Reuse in the proposed method is based on use case based queries applied to a reuse repository, organized around use case patterns [11, 12].

2 The Method - Software Factories Driven by Requirements

When developing a software system we often realize that we have already built a similar system before. We might also know that someone else (eg. in another department) have already developed something similar. When making changes or extensions to an existing system we often struggle with finding places in code where changes or additions should be made. These changes and extensions are always associated with new requirements, and most often we have long forgotten the relationships between the old requirements and the resulting code.

The above common problems are associated with lack of knowledge about the previously built systems. This knowledge is usually gone with the developers that left, or simply forgotten. All we have are vague requirements "specifications", some outdated drawings representing the system's structure and of course the code. In some cases we might even have some descriptions of the system's dynamics (like test cases or some interaction diagrams). Unfortunately, most often, the specifications have very weak links with the implementation artifacts (like code) and are very hard to reuse. Moreover, the organizations that enforce strict and constant synchronization of analysis and design artifacts with code usually fall into a disease (see [13]) that is often fatal for their projects.

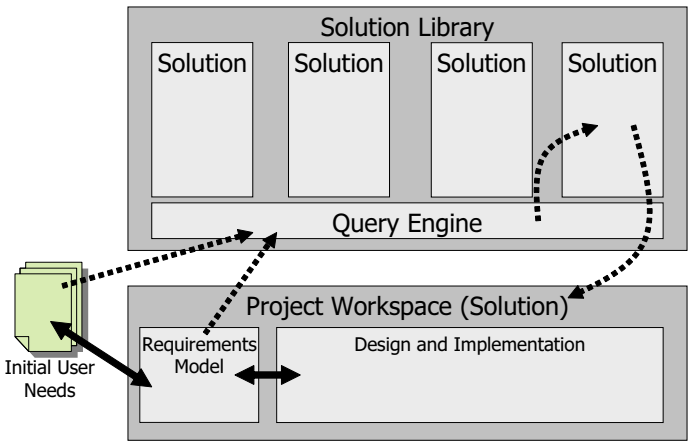

Fig. 1. Method for gathering and applying knowledge on software development

It is obvious that reusing knowledge from previous projects speeds the development process. Unfortunately, this idea seems to fail so far [14]. It can be argued that our inability to implement reuse mechanisms is due to complexity of software systems which results in problems in gathering and implementing knowledge about the development process and its artifacts. We thus need efficient mechanisms to capture knowledge while we develop software and easy to apply mechanisms that enable reuse of the previously captured knowledge.

Figure 1 shows a proposition of a method that supports the process of gathering and reusing knowledge about the software development process and its products. This method can be treated as a software factory schema in terms of [1]. In our method, knowledge is captured through artifacts (models and code) and transformations between them. We also capture knowledge about the artifact and transformation schemas (metamodels).

Knowledge about a given software development project is captured in the form of a "Solution"[2] (Fig. 1). Solutions gathered from previous projects are kept in a Solution Library. The current Solution is developed in the current Project's Workspace. Solutions are sought for in the library through a Query Engine. The most important feature of the query mechanism is that it is based on the requirements model. It is the requirements that drive the reuse process. We will explain this mechanism further on in the paper.

The structure and usage of Solutions is explained in more detail on Figure 2. Every solution (including those still developed in the project workspaces) is composed of several models that lead from the requirements to code and mappings (traces) that determine how the models are transformed. The structure of these

[2] The term 'solution' used in this paper should not be mistaken with 'solutions' used eg. in .NET.

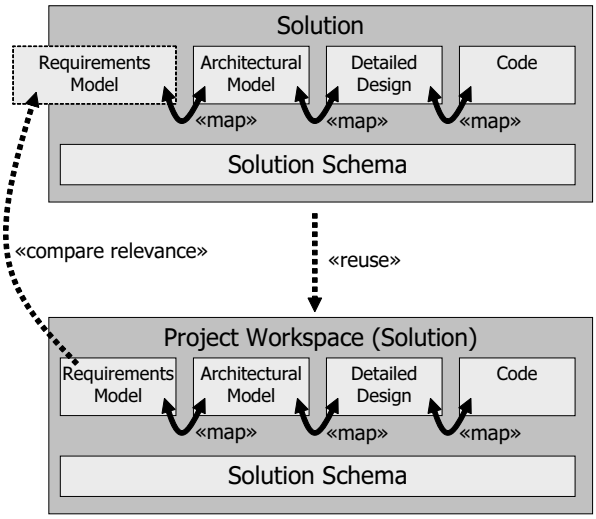

Fig. 2. Structure of solutions and their usage

models and the mappings between them are described in a Solution Schema. In order to reuse a solution, we need to formulate a query that allows for finding the most relevant solutions. We compare solutions by comparing their Requirements Models and determining their relevancy by finding the level of analogy (see [15, 16]). After finding solutions relevant through similar requirements we can reuse their contents in the current solution workspace.

The process of comparing and reusing solutions is presented on Figure 3. It is driven with three schemas (metamodels), two of which are part of the solution schema. The initial requirements model is compared with the library requirements models («compare relevance»). This comparison is done with the use of a Query Schema which describes the structure and method of application for queries performed through the Query Engine. Relevant requirements models can be reused («reuse») together with associated design and implementation artifacts (here represented only by the Architectural Model). For the models in the solutions to be developed, reused and modified, we need to define appropriate Solution Schema. This schema contains the Model Schema which defines all the model well-formedness rules and the Mapping Schema which determines the rules of model transformation.

It can be noted that the key to the presented method is the definition of the requirements model. We need a model schema that would make this model comprehensible for people and at the same time appropriate for automatic querying. Moreover, this model should be easily transformable into design models with the help of automatic tools. In the following three section we will present three schemas that define the requirements model, its mapping into other models and query application.

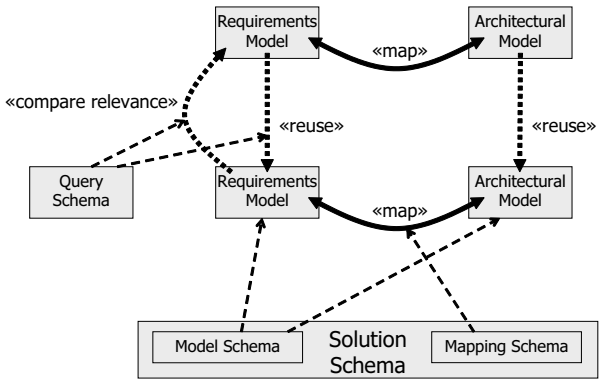

Fig. 3. Comparison and reuse process driven by schemas

3 The Schemas - How to Drive Software Factories with Use Cases

According to Figure 3 we need three schemas in order to organize our software factory. These schemas would define the requirements model and its relationships with other models. We shall base the schema for requirements on the widely used use case model. It is important that this schema should define the structure of individual use cases. This would allow for treating use cases as drivers for reuse (see [11] for some insight) and as precise specifications and units of system development (see [17] for more detailed discussion). At the same time, the schema would need to accommodate precise specification with informality required by the users that verify use cases [18].

Our model schema for requirements extends the schema for use cases (use case metamodel found eg. in [19]) with scenarios and scenario sentences which is illustrated on Figure 4. Every use case has several scenarios. A scenario consists of several ordered sentences. Sentences are formed of a subject, a verb and an object (a second object is also allowed). With this approach we precisely describe the functional aspects of a system without going into details about the vocabulary of our problem domain. The vocabulary is defined elsewhere and supplies us with all the definitions of notions used as subjects, verbs and objects in scenario sentences. More detailed discussion on this can be found in [18].

It can be noted that use cases with scenarios and scenario sentences are good means to formulate patterns on the requirements level. Such patterns are more detailed than use case patterns as described in [12]. Use case patterns offer very general guidelines on structuring the functional requirements models. In order to drive reuse and software development in a software factory we need more detailed "driver" patterns. A complete pattern should contain details about the actual use case contents, and give precise links to the design and implementation

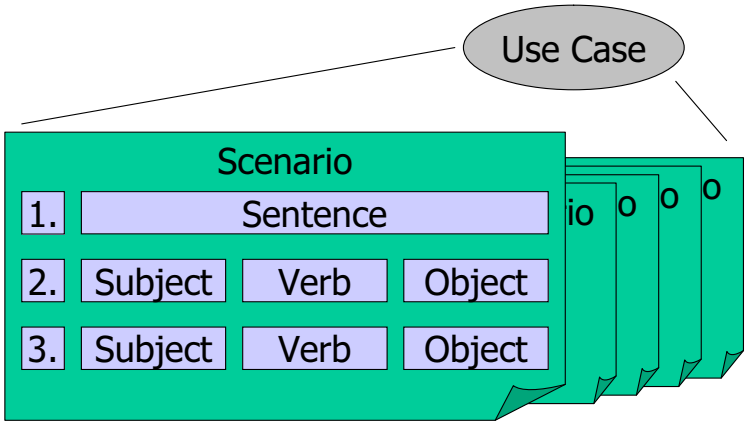

Fig. 4. Model schema for requirements based on use cases

artifacts. Patterns based on the schema illustrated on Figure 4 offer such level of detail. We can call them **scenario patterns**.

The use case model structured as described above and associated with a separate vocabulary of notions is a good basis for defining the query schema. This schema would allow us to seek for solutions (in terms of Figure 1) represented through appropriate scenario patterns. The schema, illustrated on Figure 5, defines a method for comparing relevance of scenarios in two areas - functional and domain.

Functional relevance is based on comparing subjects and verbs in sentences of one scenario with subjects and verbs of sentences in another scenario. Determining functional relevance for two scenarios means comparing subject-verb tuples in both scenarios and comparing the sequence of these tuples. Calculating relevance means counting how many subject-verb tuples in sentences are the same in both scenarios and counting how many tuple sequences are repeated. For example, the two scenarios presented below have high functional relevance. All the subject-verb tuples are the same in both scenarios. Moreover, most tuple sequences are repeated. The only difference is in the order of tuples between sentences 5 and 6. It has to be noted that the name of the subject that is associated with an actor is irrelevant for comparison. Sentences are treated as relevant if the two subjects are either actor names or both are "system".

```
Scenario 1
1. Administrator chooses to register user data.
2. System shows user data entry screen.
3. Administrator enters user data.
4. System verifies user data.
5. System shows success screen.
6. System remembers user data.
```

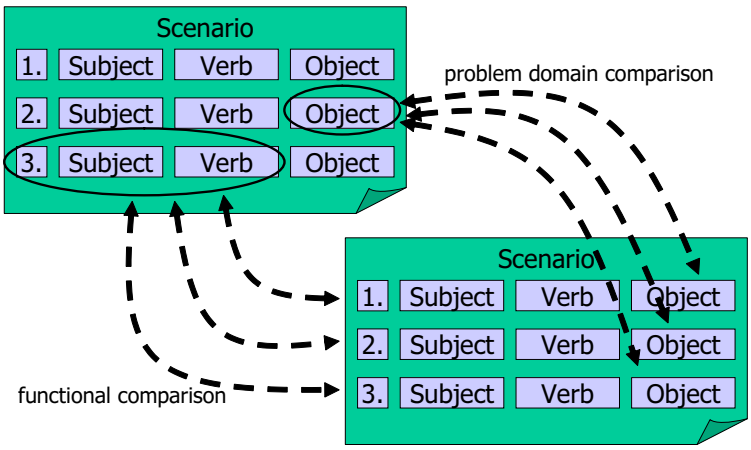

Fig. 5. Query schema allowing for scenario comparison

```
Scenario 2
1. Store-keeper chooses to register hardware data.
2. System shows hardware data form.
3. Store-keeper enters hardware data.
4. System verifies hardware data.
5. System remembers hardware data.
6. System shows acknowledgement.
```

Problem domain relevance of scenarios lies not in the sequence of scenario sentences but in vocabulary notions used therein. We thus compare sentence objects (see Fig. 5). We calculate this kind of relevance by counting the number of matching objects in two scenarios. Under this criterion, the two scenarios presented above (Scenario 1 and Scenario 2) have very poor problem domain relevance. However, Scenario 3 presented below has some degree of domain relevance with Scenario 2. They have two notions (sentence objects) in common ('hardware data' and 'hardware data form').

```
Scenario 3
1. Manager chooses to browse hardware data.
2. System shows hardware query form.
3. Manager enters hardware query.
4. System shows hardware list.
5. Manager picks hardware list entry.
6. System shows hardware data form.
```

It has to be stressed that in order to determine domain relevance it is not satisfactory to compare sentence objects literally. We have to consider the pos-

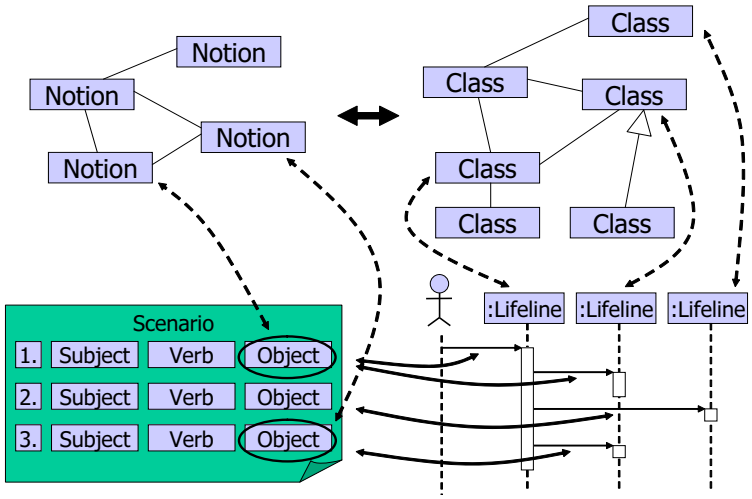

Fig. 6. Mapping schema for transformation between requirements and design

sibility of using synonyms and homonyms. This means that the organization of problem domain vocabulary is very important. The writers of scenarios have to be able to connect the objects in scenarios to appropriate vocabulary entries (associating sentence objects with problem domain notions). This mapping of scenario sentences and notions is illustrated on Figure 6. It is worth noting that keeping this mapping coherent necessitates the use of appropriate scenario construction tools (see [20] for a description).

Use case scenarios with appropriately associated vocabulary notions are the basis for constructing a reuse mechanism. In order to reuse a solution in a library we need to calculate its relevance to the current project workspace solution (see Fig. 2). Solution relevance is equivalent to relevance of their requirements models. Relevance of requirements models is a sum of relevance of individual use cases, and more specifically - functional and domain relevance of their scenarios.

Solutions found in the solution library are ordered by their relevance to the current project workspace. The developers can now reuse elements of the solution. Most relevant use cases can be reused directly. Reusing use cases means also the possibility to reuse all the models derived from the use case model. This of course is possible only when the solutions have precisely defined mappings between models. Such a mapping is illustrated on Figure 6. Scenario sentences are mapped onto messages on sequence diagrams. Vocabulary notions are mapped onto class or component diagrams. For very relevant use cases we can reuse the associated (mapped) design diagrams in their entirety. Design diagrams for less relevant use cases need some development effort in order to accommodate for changed functionality or problem domain description.

4 Conclusions and Future Work

The presented method allows for organizing the process of capturing and reusing knowledge about solutions applied when developing software systems. This knowledge is captured by means of various models originating in a use case model described through precisely defined scenarios with vocabulary notions.

In order to apply the presented method efficiently we need to fulfill several prerequisites. The most important prerequisite is a decision to use uniform requirements specification schema throughout various software development projects. Another important condition is the use of standard modeling notation (eg. UML) for denoting all the models transformed from the requirements model (up to the resulting code). Solutions based on standard model and transformation schemas can be stored in libraries maintained by software development organizations. Initiation of such a library would be a strategic decision for an organization.

It can be argued that the main obstacle for applying the presented (and any other) method for global reuse of knowledge in software engineering is lack of appropriate tools. The current efforts should thus go into the direction of developing efficient query and transformation engines. We should also develop tools to maintain scenario patterns which play key role in identifying solutions for reuse (see [20]).

Having such tools we can organize software factories based on reuse of knowledge. This knowledge allows us to reduce our efforts to develop systems similar to those already produced. It can be noted that the application of the proposed method suppresses the need to perform variability (similarity) analysis as it is done for software product lines [21]. It can be argued that the variability of user's needs is maintained by capturing and storing similar solutions in a solution library. Several such solutions form a family of systems for which we have ready knowledge to implement them in highly reduced lifecycle times. Every new similar solution extends the level of variability of a given software product line based on the solution library.

References

1. Greenfield, J., Short, K.: Software Factories. Assembling Applications with Patterns, Models, Frameworks and Tools. Wiley, Indianapolis, Indiana (2004)
2. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley (2002)
3. Miller, J., Mukerji, J., eds.: MDA Guide Version 1.0.1, omg/03-06-01. Object Management Group (2003)
4. Kruchten, P.: The Rational Unified Process: An Introduction, 3rd ed. Addison Wesley (2003)
5. Cockburn, A.: Agile Software Development. Addison-Wesley (2002)
6. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (2000)

7. Sutcliffe, A.G., Maiden, N.A.M.: Bridging the requirements gap: policies, goals and domains. In: IWSSD '93: Proceedings of the 7th international workshop on Software specification and design, Los Alamitos, CA, USA, IEEE Computer Society Press (1993) 52–55
8. Hjelm, T.: Closing the requirements gap with unambiguous models. RTC **12** (2003) – http://www.rtcmagazine.com/home/article.php?id=100223.
9. Rosenberg, D., Scott, K.: Use Case Driven Object Modeling with UML. Addison Wesley (1999)
10. Laguna, M.A., López, O., Crespo, Y.: Reuse, standardization, and transformation of requirements. Lecture Notes in Computer Science **3107** (2004) 329–338
11. Śmiałek, M.: Global reuse strategy based on use cases. In: OOPSLA 2000 Companion, Conference on Object-Oriented Programming, Systems, Languages and Applications, Minneapolis (2000) 49–50
12. Overgaard, G., Palmkvist, K.: Use Cases: Patterns and Blueprints. Addison Wesley (2005)
13. Bell, A.E.: Death by UML fever. Queue **2** (1) (2004) 72–80
14. Schmidt, D.C.: Why software reuse has failed and how to make it work for you. C++ Report **11** (1) (1999) – http://www.cs.wustl.edu/ schmidt/reuse-lessons.html.
15. Pan, Y., Wang, L., Zhang, L., Xie, B., Yang, F.: Relevancy based semantic interoperation of reuse repositories. In: SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, New York, NY, USA, ACM Press (2004) 211–220
16. Hamza, H., Fayad, M.E.: Applying analysis patterns through analogy: Problems and solutions. Jornal of Object Technology **3** (4) (2004) 197–208
17. Śmiałek, M.: From user stories to code in one day? Lecture Notes in Computer Science **3556** (2005) 38–47
18. Śmiałek, M.: Accommodating informality with necessary precision in use case scenarios. Journal of Object Technology **4** (6) (2005) 59–67 http://www.jot.fm/issues/issue_2005_08/article5.
19. Object Management Group: Unified Modeling Language: Superstructure, version 2.0, Revised Final Adopted Specification, ptc/04-10-02. (2004)
20. Śmiałek, M., Bojarski, J., Nowakowski, W., Straszak, T.: Writing coherent user stories with tool support. Lecture Notes in Computer Science **3556** (2005) 247–250
21. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison Wesley (2004)