Integrating Software Lifecycle Models into a uniform Software Engineering Model

Jonas Helming

Technische Universitaet Muenchen
Department of Computer Science
Chair for Applied Software Engineering
Bolzmannstraße 3
D-85748 Garching bei München, Germany
helming@in.tum.de

Abstract: Software lifecycle models differ in their artifacts as well as in the dependencies between the included tasks and activities. Therefore support-tools, which support these lifecycles, are typically specialized on a certain lifecycle. A tool for Scrum [Sc07] for example may model user stories and sprints, but exclude risksless shift of the applied software lifecycle, e.g. from a heavyweight to an agile methodology. Furthermore existing tools are mostly isolated from the product model, which complicates traceability between modeling tasks and the objects of tasks, e.g. UML models [O04]. This paper proposes a unified Meta-Model to integrate the software lifecycle model into the Rational-based Uniform Software Engineering model (RUSE) [Wo07]. The meta-model aims at two goals, and further evaluation is required to determine to which degree they are satisfied. First the integration between process and product model should enable a complete traceability from tasks to their objects. Second in the meta-model every artifact in the project can be potentially part of the process model, which should allow to seamlessly shift the applied software lifecycle.

1 Introduction

Triggered by the still ongoing debate between traditional and agile approaches, the search space for choosing a software lifecycle model has been expanding. The recommended practice is to select and tailor a particular methodology for a software project according to specific criteria. Glass, for example considers the following four criteria [Gl02]: size, application domain, criticality and innovativeness. Agile methods like Scrum [Sc07] or XP [Be00] are targeted at small to medium sized projects, whereas highly critical projects tend to be managed by risk-oriented methodologies like the Spiral model [Bo88]. If the values of these criteria are considered to be constant for the duration of the project (e.g. [KV00]), the choice of a software lifecycle model can be made at the beginning of a project. But we believe these criteria cannot be considered as constant, and therefore that a shift of the software lifecycle model must be supported. We propose a meta-model, which integrates the software lifecycle model into the Rational-based Uniform Software Engineering model (RUSE) [Wo07]. As a consequence tasks become completely traceable from their origin to their objects. Using

this meta-model every artifact and the associations between them can be part of the process model. We claim this allows modeling every software lifecycle. This is a precondition for a tool-supported shift of the software lifecycle model. Further techniques to handle change known from the field of product modeling like software configuration management can also be applied to the process model.

2 The unified Meta-Model

This section describes the unified meta-model, which integrates process modeling into the existing Rational-based Uniform Software Engineering model (RUSE) [Wo07]. RUSE supports distributed software engineering projects in system modeling, collaboration and organization. The model is implemented in a tool suite called Sysiphus [Br06]. The key idea of RUSE is the combination of different software engineering models into a unified model. Every element in this unified model is a ModelElement. The RUSE model supports three different categories of ModelElements: SystemElement, CollaborationElement and OrganizationElement. SystemElements are used to model the system under construction on different layers of abstraction. Most of the SystemElements are based on the Unified Modeling Language (UML) [O04], but RUSE also supports other abstractions like the UserStory used in XP [Be00]. CollaborationElements capture the communication and collaboration of users and are mainly based on the QOC [Ma96] model. CollaborationElements range from comments, action items, and risks to milestones and iterations. OrganizationElements describe the organizational structure of a software development project and are used to model organizational units such as teams and participants and their associations. Associations between different types of ModelElements are modeled as links between them. Links can relate two model elements inside a model diagram, for example two classes in a UML class diagram. But links can also relate two model elements from different diagrams, for example a link connecting a developer to a subsystem he is working on. RUSE is extensible, that is, new types of ModelElement and links can be added to the meta-model. For example, a new software process can be introduced by adding new model elements. To support Scrum [Sc07], a class Sprint has been added [Sc06]. This extensibility makes RUSE the ideal basis for integrating different software lifecycles. One could be tempted to add a new category for every software lifecycle, but the challenge is to interrelate different existing (and emerging) process models within RUSE. This is a precondition to shift between different software lifecycle models at “project runtime”. Therefore we define a meta-model for RUSE extracting the commonality of process models.

In RUSE only a few ModelElements like Issue or ActionItem have status information. Different methodologies use a variety of elements to represent the status of a project. We distinguish two different types: First elements with an explicit status, which can be declared and second elements with an implicit status, which is dependent on other elements. Typical examples of the first type are checklist items like “Sprintlog Items” in Scrum [Sc07] or “Risks” in the Spiral model [Bo88]. An example for the second type is a phase, which is dependent on the status of its containing activities. As the unified

meta-model must be able to model every methodology we extend the RUSE meta-model: Every ModelElement can now have a status, which can be observed with a specific perspective. In addition to the two states "open" and "closed", we add the status "blocked" to indicate the dependency of an element on the completion of another element. We therefore extend the RUSE ModelElement with an attribute, which can be closed, open or blocked. We call this attribute COB (as an acronym for Closed Open Blocked). The acronym helps us to distinguish the new attribute from already existing attributes like status or state, which are already implemented by several subtypes of ModelElement (like the Issue). The meta-model allows, that the status of elements is dependent on other elements. Therefore the meta-model has to extract the communality of the dependencies used in different methodologies. There are types of dependencies called BlockingDependency and OpeningDependency. BlockingDependencies are used in activities that require sequential execution, for example the activities in the waterfall model are connected by BlockingDependencys. An example for an OpeningDependency in a Scrum-based project is the association between SprintBacklog and SprintBacklogItems. The status of the SprintBacklog is depending on the status of the included SprintBacklogItems. The SprintBacklog is open as long as there are open SprintBacklogItems in it. In the following we will define the effect of BlockingDependencies and OpeningDependencies in the meta-model. Dependencies connect two ModelElements. If two ModelElements are connected by a BlockingDependency we call one of them Blocker and the other Blocked. If the dependency is an OpeningDependency, we call the ModelElements Opener and the other Opened (see Figure 2). Opener and Blocker are also called Source, and the term target is used for Opened and Blocked. If the source of a dependency is open or blocked we call the dependency active. The COB of a Blocked and an Opened is affected by these dependencies if they are targets of active dependencies:

- One or more active OpeningDependency: The Opened is open.
- One or more active BlockingDependency: The Blocked is blocked.
- Both: The target is blocked.

In the meta-model the dependencies are not restricted to specific elements, as every element can be part of the process. Therefore even dependencies from the system model can be used for process modeling. Figure 1 shows an example for the effect of dependencies between various elements. A Task is connected to an Activity with an OpeningDependency as the Activity is open until all included tasks are closed. As long as this Task is open, the dependency is active. The Task also opens a Subsystem, the object of the task. This Subsystem is connected to second Subsystem by a BlockingDependency, because the second Subsystem depends on it. As this dependency is also active, the second Subsystem is blocked.

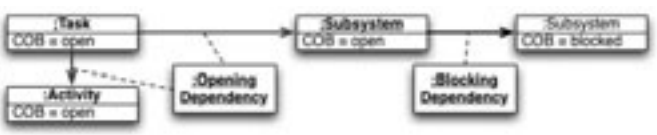

Figure 1: Example for the effect of dependencies

ModelElements in RUSE are connected by ModelLinks. There is a variety of existing types of ModelLinks, e.g. issues can be connected to sub issues. To integrate the dependencies in RUSE we don't necessarily introduce new ModelLinks, but we classify the existing types. Every ModelLink can either be an OpeningDependency, a BlockingDependency or NoDependency (default). To model Scrum, for example, we classify the link between a sprint backlog item and a sprint backlog as an OpeningDependency, with the sprint backlog item as source. As a consequence, the sprint backlog will be open as long as any of its sprint backlog items is open or blocked.

ModelElements are closed as long as they are not target of an active dependency. Also it is not allowed to change the COB explicitly. Therefore we need another type of element to indicate that there is open work to do in a project. We call these elements Checkables, the smallest units in the work breakdown structure of a project. The complexity of Checkables should be on a level, where project members are able to decide explicitly if they are closed. Checkable is a subclass of model element (see Figure 2) inheriting the attribute COB and the two types of dependencies. Checkable extends ModelElement by the attribute checked which is manipulated by the methods check() and uncheck(). The attribute checked affects the COB of the checkable in the following way: If a Checkable is unchecked and is not target of active dependencies it is open. In any other case the Checkable COB is handled the same way as in the super class ModelElement. Typical examples for Checkables in RUSE are ActionItems, Issues or Risks.

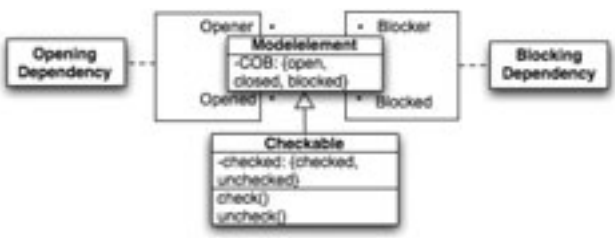

Figure 2: The meta-model

3 Examples for using the meta-model

In this section we will describe some examples how existing software lifecycle models can be modeled as instances of the introduced meta-model. We start with an activity-oriented perspective of a sequential waterfall model consisting of the activities RequirementsAnalysis, SystemDesign and ObjectDesign (see Figure 3)[1]. The activities block each other, thus they are linked by a BlockingDependency. Every activity results in a corresponding document. An OpeningDependency connects the activities to their

[1] In the figures we use highlighting schema to indicate the value of the COB attribute: Closed ModelElements are shown as white boxes, open instances are drawn in grey and blocked instances are are shown in black. Furthermore we highlight OpeningDependencies in grey and BlockingDependencies in black. Boxes with numbers denote arbitrary ModelElements.

respective documents. This implies, that the documents are open until the corresponding activity is closed. In this example, the SystemDesign activity and the ObjectDesign activity are still blocked. In the RequirementsAnalysis activity some work still needs to be done (see items “1”, “2” and “3”). These ModelElements are connected to the activity by an OpeningDependency. Therefore the activity remains open until all these ModelElements are closed.

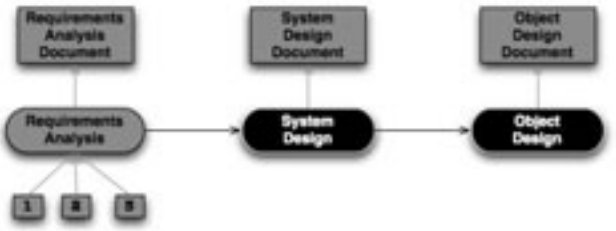

Figure 3: Waterfall model

In the next example (see Figure 4) we model a project using a Scrum process. We introduce more ModelElements to represent a progressed project. Also we introduce dependencies between these ModelElements, e.g. “3” blocks “5”. For example this can represent, that a component of a system cannot be realized until another component is finished. The Repository contains all ModelElements. The ProductBacklog in Scrum contains all tasks of a project over time. We model this by connecting every Modelelement in the repository to the ProductBacklog by an OpeningDependency. The ProductBacklog perspective shows all open ModelElements of a project, which are not yet connected to a sprint. We introduce two sprints and connect them by a BlockingDependency. Sprint2 is blocked until Sprint1 is closed. To plan sprints we connect selected ModelElements to the according sprint by an OpeningDependency. The event-oriented perspective shows the open elements of every sprint.

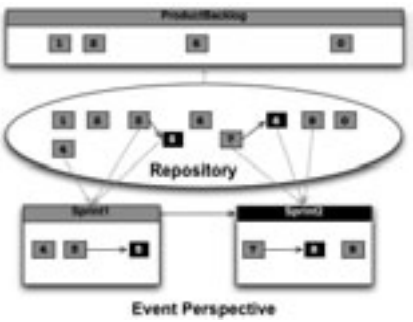

Figure 4: An instance of the meta-model modeling Scrum

Processes are modeled by connecting ModelElements in the repository to elements like activities, sprints or documents. This allows for developers to apply more than one process in the same project, even if they are working on the same repository. We demonstrate this in Figure 5: We use two processes, the waterfall model from the first example and a Scrum process. A developer can now seamlessly switch between Scrum and the waterfall process model since they are only two different perspectives on the same repository. This allows for interesting combinations. For example, the project can be internally managed with Scrum, while the manager can present the status of the project using the waterfall model perspective to a client. In Figure 5 all ModelElements of Sprint1 are already closed, which means that Sprint1 is also closed. Therefore Sprint2 is not blocked anymore, but open.

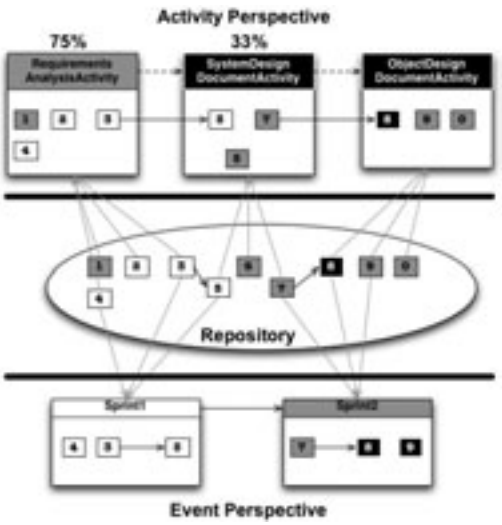

Figure 5: Modeling waterfall model and Scrum based on the same repository

5 Future work

The unified meta-model is currently being implemented in the Sysiphus tool-suite. Our next step is the creation of model-templates for the most common methodologies and perspectives. We already have experiences with projects, which evolutionary tailored their process from a traditional to an agile process. But this was done without the support of a process model like we suggest in this paper and without explicit tool-support. Beginning in October 2007 we have done an empirical evaluation of our ideas using a case study based on a large student project with an industrial partner. We will try to model a process dynamically retailored from a waterfall model to a Scrum oriented approach over the project duration.

Bibliography

[Be00] Beck, K.: Extreme Programming Explained, Addison-Wesley, 2000.

[Bo88] Boehm Barry W. A Spiral Model of Software Development and Enhancement, IEEE Computer, IEEE, New York, Mai 1988

[Br06] B. Bruegge, A. H. Dutoit, and T. Wolf. Sysiphus: Enabling in formal collaboration in global software development. In Proceedings of the First International Conference on Global Software Engineering, October 2006.

[Gl02] Glass, Robert L. "Agile Versus Traditional: Make Love, Not War!" in "The Great Methodologies Debate: Part 1", *Cutter IT Journal*, Vol. 15 No. 1 (January 2002)

[KV00] Khalifa M. and Verner June. Drivers for Software Development Method Usage, IEEE Transactions on engineering management, IEEE, New York, August 2000

[Ma96] MacLean A., Young R.M., Bellotti V. and Moran T. "Questions, options and criteria: Elements of design space analysis", Human-Computer Interaction, Vol. 6, 1996.

[O04] OMG Unified Modeling Language Specification Version 2.0, May 2004.

[Sc07] Scrum Alliance, URL: http://www.scrumalliance.org, 1. September 2007

[Sc06] Schiller, Jennifer. *Scrum in the unified project model*. Diploma thesis, Technische Universität München, September 2006

[Wo07] Timo Wolf. *Rational-based Unified Engineering Model*. Dissertation, Technische Universität München, July 2007