Framework for Describing UML Compatible Development Processes

Pavel Hruby

Navision Software
Frydenlunds Allé 6
DK-2950 Vedbæk, Denmark
ph@navision.com

Abstract. Have you ever tried to specify an accurate development process for your organization and later faced difficulties with the complexity of the description? Instead of describing a specific process, it might help to describe a process framework and reuse it by creating specific processes for specific needs. This paper describes the object-oriented framework of a development process, which considers software development artifacts as objects and evolution as collaborations between the objects. Such an object-oriented process definition can deal with the complexity of a development process in a better way than a traditional description based on workflow. This paper discusses features of such a process framework with an eye towards approaches such as Fusion, OPEN and the Rational Unified Process.

Acknowledgement. I would like to thank to Prof. B. Henderson-Sellers of the School of Computing Sciences, University of Technology, Sydney, Australia, for "shepherding" the article, and for his useful suggestions and comments. I do, of course, take full responsibility for any omission or errors.

1 Introduction

"The software development process, as actually performed, is so complex that we cannot write it down accurately, and if we could, no one could read that description and learn to perform it," (Alistair Cockburn, panel discussion, ECOOP'98 [2]). In this paper, this problem is solved in the following way. Instead of writing down a concrete process scenario, we specify a process framework that describes all allowable processes. Such a framework is abstract yet precise. To meet the demands of specific development problems, the framework is reused by creating specific development processes. The concrete development processes can be represented at the necessary level of accuracy. This solution significantly simplifies the description of concrete development processes, because the complexity is localized in the abstract form in the process framework.

This paper is structured in the following way. The second section explains the traditional specification of software development processes and its drawbacks. The next four sections explain the main ideas of the object-oriented process specification, and the structure of the object-oriented framework is outlined in the section 6: The Object-Oriented Specification of Development Processes. The next section compares the object-oriented specification with the original specification of three contemporary design methods and methodological frameworks.

2 Traditional Specification of Software Development Processes

The purpose of this section is to clarify the terminology used throughout the paper because different authors define these terms differently in different contexts.

The traditional specification of a development process is typically illustrated with a graph of tasks, techniques, software development artifacts and activities. *Tasks* are small behavioral units that usually result in a software development artifact. Examples of tasks are construction of a use case model, construction of a class model and writing of code. *Techniques* are formulas for performing tasks, for example, object design using CRC cards, functional decomposition and programming in Visual Basic. *Software development artifacts* are final or intermediate products resulting from software development, for example, a use case model, a class model, or source code. *Activities* (in this paper) are units that are larger than task units. Activities typically include several tasks and software development artifacts. Examples of activities are requirement analysis, logical design and implementation.

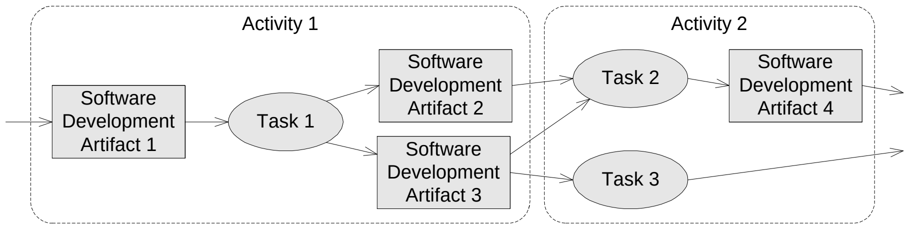
Activity 1
Activity 2
Software Development Artifact 1
Task 1
Software Development Artifact 2
Software Development Artifact 3
Task 2
Task 3
Software Development Artifact 4

Fig. 1. Traditional specification of a development process. It can be compared with the object-oriented specification illustrated in Fig. 10

In general, the traditional specification of a development process cannot cover all the possible combinations of activities and software development artifacts without becoming overly complex. This paper will show that in the specification of development processes, the object-oriented approach deals with complexity of the development process better than the traditional approach illustrated in Fig. 1. This is similar to experience from within software development, where the object-oriented approach can deal with the complexity of large software solutions better than the traditional structured approach.

3 Basic Features of the Product-Focused Object-Oriented Process Specification

Software development and management artifacts produced during a software development process are considered objects with various methods and attributes. Evolution during software development is represented as collaborations between software development artifacts, management artifacts and users of the method.

The object-oriented specification of the software development process distinguishes between artifacts and their representations. A *software development artifact* determines information about a *software product*. Examples of software development artifacts are use cases, software architecture, object collaborations and class descriptions. A *management artifact* determines information about a *management product*, such as a project and a team. Software development artifacts can be very abstract, such as the vision for the software system, or very concrete, such as the source code. The *representation* determines how the artifact is presented. For example, a use case model is represented by a use case diagram; a state model can be represented by a statechart diagram, an activity diagram or a state transition table. The object interaction model can be represented by a set of sequence diagrams or a set of collaboration diagrams. Various design methods typically recommend a suitable representation of each software development artifact. However, the choice often depends on the concrete situation, and it is sometimes advisable to leave the final decision about the representation to a user of the method.

An artifact has a representation, properties, responsibilities, methods, relationships to other artifacts and attributes, all of which are discussed later. Consistency check, process phase or technique, for example, are not called software development or management artifacts in this article because they do not describe design of a software or management *product.*

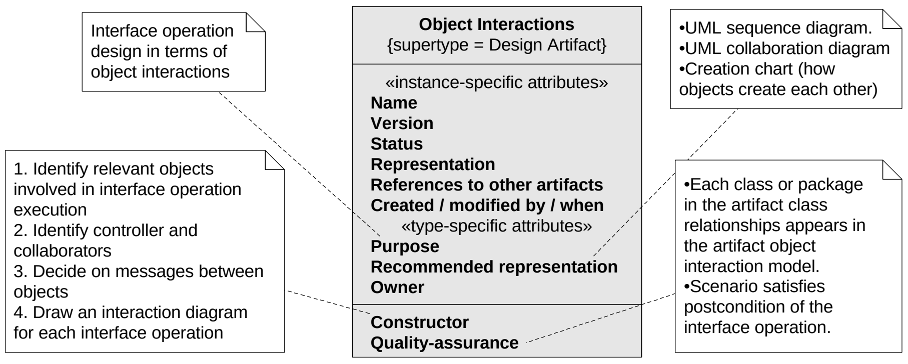
Interface operation design in terms of object interactions
1. Identify relevant objects involved in interface operation execution
2. Identify controller and collaborators
3. Decide on messages between objects
4. Draw an interaction diagram for each interface operation
Object Interactions
{supertype = Design Artifact}
«instance-specific attributes»
Name
Version
Status
Representation
References to other artifacts
Created / modified by / when
«type-specific attributes»
Purpose
Recommended representation
Owner
Constructor
Quality-assurance
•UML sequence diagram.
•UML collaboration diagram
•Creation chart (how objects create each other)
•Each class or package in the artifact class relationships appears in the artifact object interaction model.
•Scenario satisfies postcondition of the interface operation.

Fig. 2. Specification of the artifact type object interaction model

The object-oriented specification of the development process distinguishes between the artifact *type* and the artifact *instance*. *Types of artifacts* specify properties, attributes and methods of various kinds of software development and management artifacts. *Instances of artifacts* are concrete software development and management products produced during software development. An example of a software development artifact type is a use case model. An example of a software development artifact instance is a concrete set of use cases, actors and their relationships, represented by a use case diagram.

Artifact types have two kinds of methods:

- *Constructors*, which are methods describing how to create an artifact.
- *Quality-assurance methods*, such as completeness and consistency checks.

Artifacts have instance-specific *attributes*: name; version; representation, which typically contains a diagram, a table or a text; status, such as draft, completed, tested; references to other software development and management artifacts; and attributes such as who created and modified the artifact and when. In addition, artifact types have type-specific *attributes*: the purpose, the recommended representation and the owner of the artifact type. Artifacts may have other additional attributes and methods than those mentioned above. Fig 2. illustrates the object-oriented specification of a software development artifact type with attributes and methods. The inheritance diagram of the artifact types is in Fig. 3.

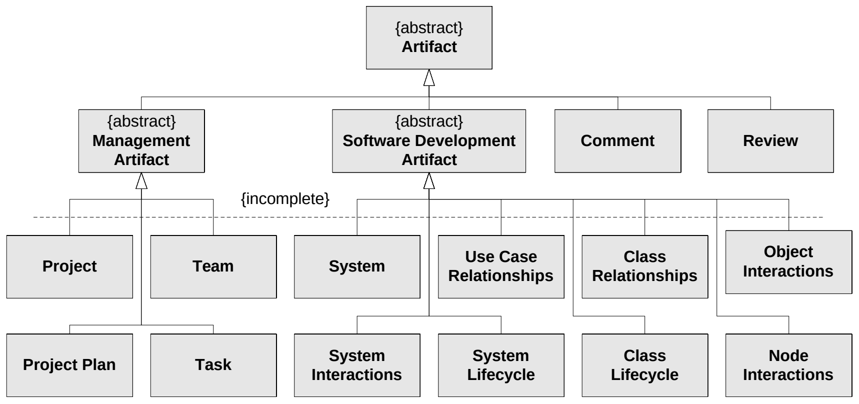

Fig. 3. The inheritance diagram illustrating types of artifacts used by the development process. The fact that the inheritance tree is incomplete allows for flexibility throughout the process and for creating new artifacts to match different kinds of development processes

4 Static Structure of Software Development and Management Artifacts

This section specifies the static relationships between software development and management artifacts. The static structure is based on the pattern for structuring project repositories with UML design artifacts [11]. This section outlines how the pattern is applied to create a static structure of the framework for describing UML compatible development processes.

A software system can be represented from various viewpoints and at various levels of granularity, see Fig 4. Examples of views[1] and levels of granularity are discussed later. In each view and at each level of granularity, a UML compatible system can be described by four artifacts: static relationships between classifiers[2], dynamic interactions between classifiers, classifier responsibilities and classifier lifecycles.

The artifact called *classifier relationships* specifies static relationships between classifiers. The artifact called *classifier interactions* specifies interactions (an instance of a scenario) between classifiers. The artifact *classifier* identifies the classifier and specifies classifier responsibilities and other static classifier properties, for example, a list of classifier operations with preconditions and postconditions, and a list of classifier attributes that can be read and set. The *classifier lifecycle* specifies classifier state machine and dynamic properties of classifier interfaces, for example, the allowable order operations and events.

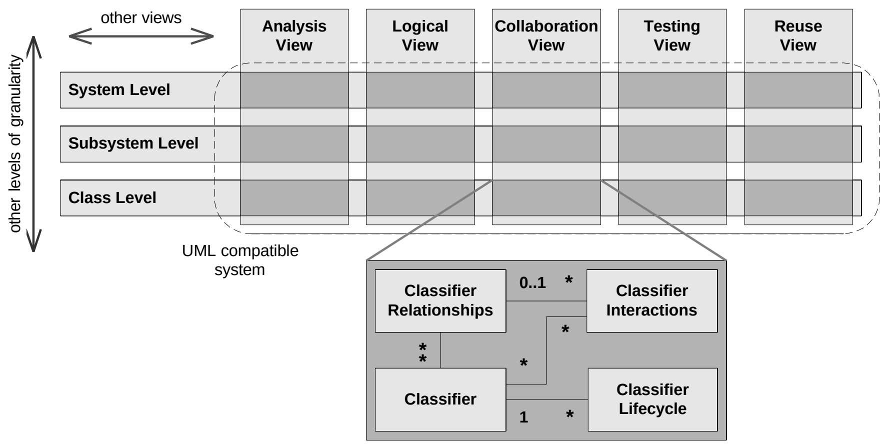

Fig. 4. At each level of granularity and in each view, the software product can be described by four types of software development artifacts. Each software development artifact identifies specific information about the software product (After ref. [10].)

Examples of views are the logical view, the collaboration view, the deployment view and the analysis view. The *logical view* describes the logical structure of the product in terms of subsystems and classes and their responsibilities, relationships and interactions. The *collaboration view* identifies types of collaborations with actors of the system, subsystems, classes, components and nodes. The *deployment view*

[1] In this article, I use the term “views” to mean complete “slices” through a model of a software system across different levels of granularity from various viewpoints. This meaning is different from the term “view”, as used in reference [12], where it means a non-complete set of significant elements.

[2] Classifiers represent static entities in a system model. In UML, classifiers are class, object, interface, datatype, use case, subsystem, component and node. Management artifacts, such as team and project, are mapped to UML as stereotyped classes.

describes the physical structure of the system in terms of hardware devices and their responsibilities, relationships and interactions. The *analysis view* describes the logical structure of the product in terms of analysis subsystems, objects and their responsibilities, relationships and interactions. The analysis view differs from other views in the way that the software entities in the analysis view do not specify the software system precisely. The purpose of the analysis view is to record preliminary or alternative solutions to design problems, and to record requirements or user's view of the system. Analysis objects may – but do not always – correspond to logical or physical software entities existing in the product.

Examples of levels of granularity are the system level, the subsystem level and the class level (see Fig. 4). The *system level* of granularity describes the context of the software system. The system level specifies the responsibilities of the system being designed and the responsibilities of the other systems that collaborate with it, responsibilities of physical devices and software modules outside the system, and static relationships, along with the dynamic interactions between them and the system being designed. The *subsystem level* of granularity describes subsystems, software modules and physical devices inside the system, along with their static relationships and dynamic interactions. The *class level* of granularity describes the detailed design of the subsystems in terms of classes and objects, and their relationships and interactions.

The software product can be represented by additional views, such as the *testing view* and the *view of reusable elements*. The software product can be specified at additional levels of granularity, such as the *tier level* for systems with layered architecture and the *organizational level* for business systems. At each additional level of granularity and in each additional view, the software product is specified by static relationships between classifiers, dynamic interactions between classifiers, classifier responsibilities and classifier lifecycles. The semantics of these additional software development artifacts are out of the scope of this paper. See paper [10], *Structuring Design Artifacts with UML*, for details about these artifacts.

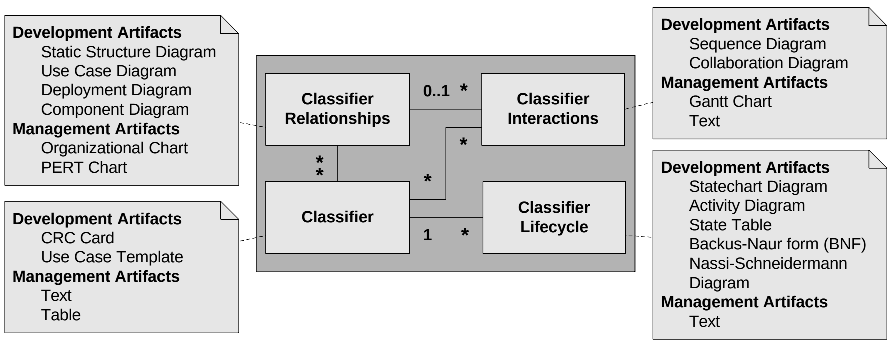

Fig. 5. Typical representation of software development and management artifacts

As mentioned in section 3, the object-oriented model distinguishes between the information itself (called software development or management artifact in this article) and its representation. Software development artifacts can be represented by UML diagrams, tables or text, see Fig. 5. The artifact *classifier relationships* is represented by a UML static structure diagram, a use case diagram, a deployment diagram and a component diagram, if classifiers are classes, use cases, nodes and components, respectively. The artifact *classifier interactions* is represented by a UML sequence diagram and a collaboration diagram. The UML Notation Guide describes only interaction diagrams in which classifiers are objects; it does not describe interaction diagrams in which classifiers are use cases, subsystems, nodes or components. These diagrams are discussed in [10]. The artifact *classifier* is represented by a CRC card, use case template, structured text or table. The artifact *classifier lifecycle* is represented by a UML statechart diagram, activity diagram, a state table, a Backus-Naur form and a Nassi-Schneidermann diagram.

Management artifacts can be represented by project diagrams, tables or text (see Fig. 5). The artifact *classifier relationships* is represented by an organizational chart, if the classifiers are roles or teams; or by a PERT chart, if the classifiers are tasks and projects. Overeager UML users can use class diagrams, in which the classes have a stereotype «task» and «project». The artifact *classifier interactions* is represented by a Gantt chart if the classifiers are projects or by text if the classifiers are roles and teams. The artifacts *classifier* and *classifier lifecycle* are represented by table or text.

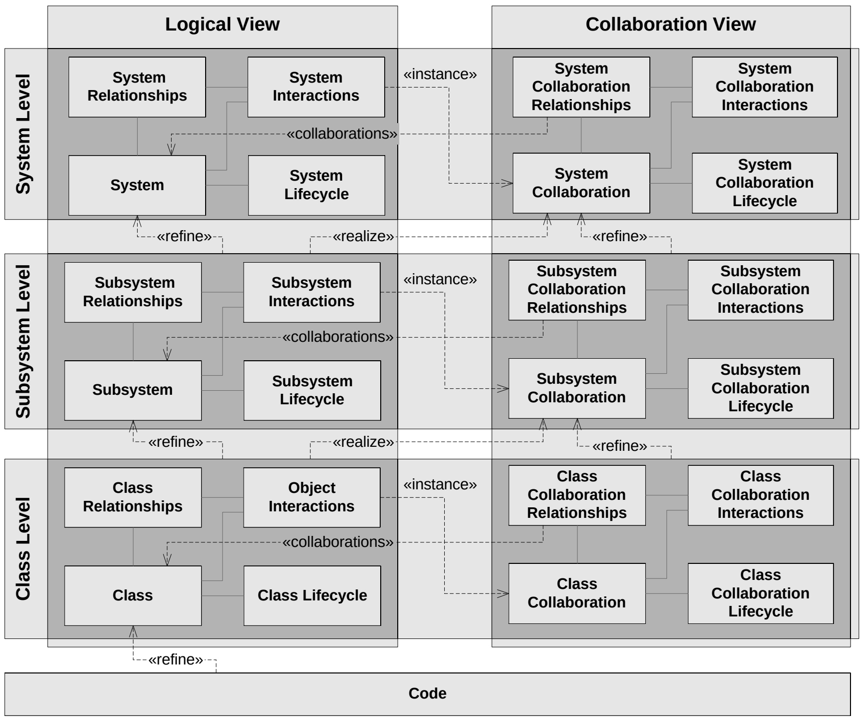

Fig. 6. Static structure of software development artifact types at the system, subsystem and class levels of granularity and in the logical and collaboration views (After ref. [10].)

Fig. 6 specifies the static structure of software development artifact types at three levels of granularity and in the logical and collaboration views. Each software development artifact type specifies certain information (discussed in paper [10]) about the software system. The structure uses the pattern described above, and the result is a regular structure, which allows consistent customization of the design specification. In simple cases, the specification consists of only a small subset of the software development artifacts identified in Figs. 5 and 6. Conversely, if software designers have to specify something unusual or unexpected, such as the things not covered by the method, the specification is extended by adding additional views and additional levels of granularity.

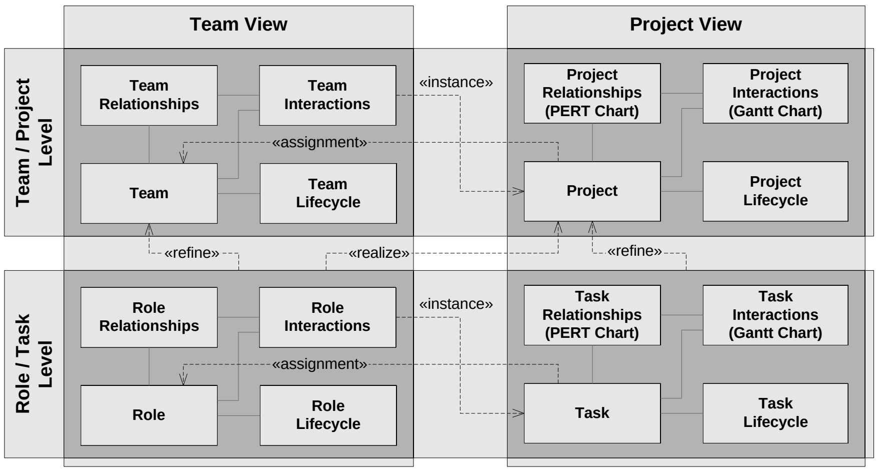

Fig. 7. Static structure of management artifact types in the team and project views

Fig. 7 illustrates the application of the pattern for structuring management artifacts. The pattern can be used to structure two kinds of management product: teams and projects. Management artifacts can be shown as stereotyped classes in UML, such as «team», «role», «project» and «task». The artifacts *team relationships* and *role relationships* specify the organizational structure at two levels of granularity. The artifact *team* specifies the responsibility of the team and the artifact *role* specifies the role of the team member. Examples of roles are developer, program manager, product manager, user education and logistics. The artifacts *team interactions* and *role interactions* specify scenarios - interactions between teams and team members, which are responses to various events. The artifacts *project* and *task* specify static properties of projects and tasks. The artifact *project relationships* and *task relationships* specify static relationships between projects and tasks. These artifacts can be represented by a PERT chart. The PERT chart shows the task dependencies, which are the most important static relationships between tasks. The artifact *task interactions* specifies a project scenario in terms of starting and finishing tasks. Accordingly, the artifact *project interactions* specifies the project scenario in terms of starting and finishing projects. These artifacts are typically represented by Gantt charts, but they might be represented by UML sequence diagrams as well. Gantt charts show the task constructors, which are the most important messages between tasks.

It can be noted that every project generates a number of artifacts not captured by the pattern. Examples of such artifacts are a glossary, minutes of meetings, reviews, comments and notes. These artifacts do not describe a *product*, and therefore they cannot be structured using the pattern. These artifacts can be related to any other software development or management artifact. For example, the glossary is related to the artifact *system*, the minutes of meetings are related to the artifact *project*. In order to reuse them in a consistent way, they have specified types in the process repository. They are illustrated, for example, in the inheritance diagram in Fig. 3.

5 Dynamics of Software Development Artifacts - Development Processes

The previous section described the static structure of software development and management artifacts. In the object-oriented specification of a development process, evolution is seen as collaborations between artifacts, and between artifacts and members of a development team. Software development artifacts can be created and completed in various orders depending on the features of various design methods. This section describes two typical examples of design processes: the Rational Unified Process and the process of the Fusion method.

Processes of different development methods create different subsets of the software development artifacts identified in the previous section, because different methods focus on different aspects of software development.

5.1 Rational Unified Process

The object-oriented specification of the Rational Unified Process [12], [15] is illustrated in Fig. 8. The figure illustrates the scenario of the requirements, analysis and design workflows of the Rational Unified Process. The figure shows the evolution of the software as a number of interactions between the worker (such as the system analyst, the use case specifier, the architect and the designer) and the software development artifacts of the Rational Unified Process.

A worker who uses the Rational Unified Process is responsible for calling the constructors of the software development artifacts in the order illustrated in the Fig. 8. Constructors and quality assurance methods generate various messages between the objects.

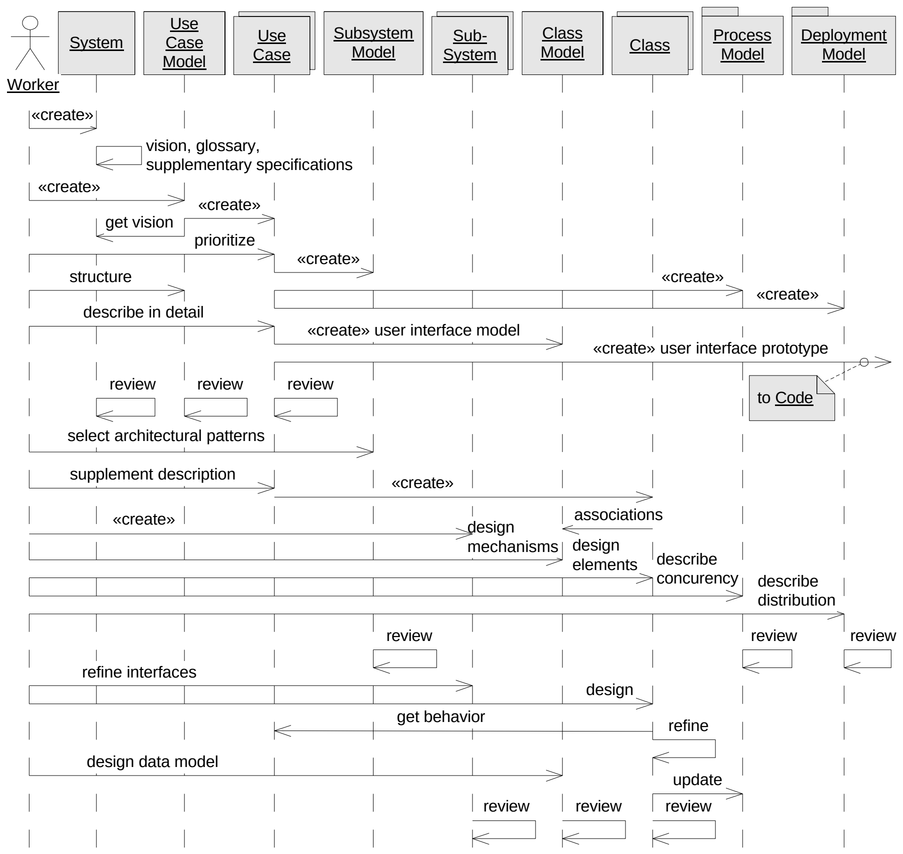

Fig. 8. The process of the requirements, analysis and design workflows of the Rational Unified Process

5.2 The Process of the Fusion Method

The object-oriented specification of the development process of the Fusion method [3] is illustrated in Fig. 9. The figure illustrates the Fusion development scenario in terms of interactions between software development artifacts, and between software development artifacts and the developer.

The Fusion method, as described in reference [3], contains more details than the Rational Unified Process 5.0, as described in reference [12]. For the sake of comparison with the Rational Unified Process (to keep the Figs. 8 and 9 at the same level of detail), Fig. 9 shows only high level interactions of the Fusion method. Note that this figure is meant as an illustration of the idea, not as a detailed description of the entire method.

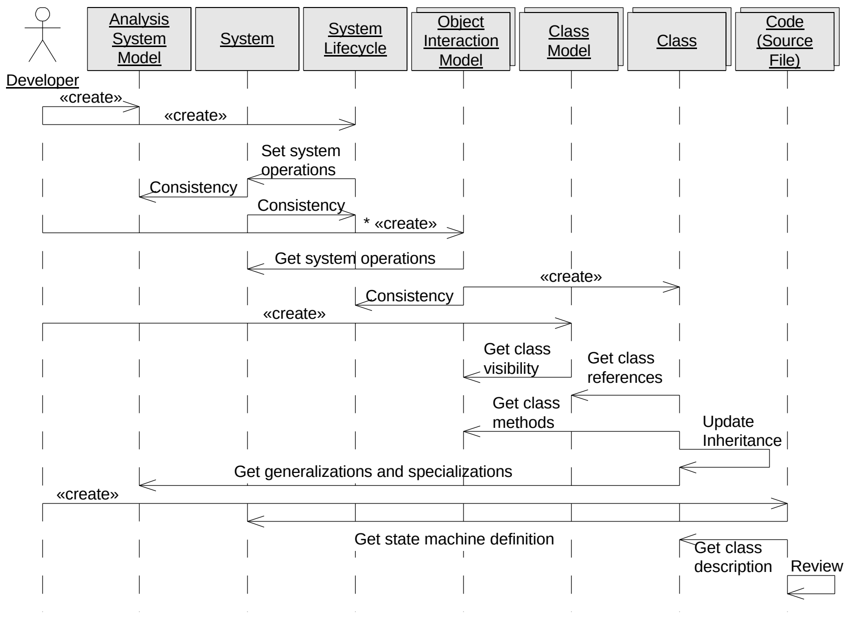

Fig. 9. The design process of the Fusion method

6 The Object-Oriented Specification of Development Processes

This section describes the framework for an object-oriented specification of development processes using the pattern of four artifacts discussed in section 4. The structure of the process specification framework is illustrated in Fig. 10.

The artifact called *artifact relationships* specifies the static relationships between software development and management artifacts. It can be represented by a UML static structure diagram, where classes and objects have stereotypes, such as «class relationships», «class lifecycle», «project» and «team». The artifact called *artifact interactions* specifies the development scenario. This artifact can be represented by a UML sequence diagram or a UML collaboration diagram. In the Rational Unified Process [12] these scenarios are called *workflows*. The artifact called *artifact type* specifies the purpose, constructor, quality assurance, and specific attributes of software development and management artifacts. The artifact called *artifact lifecycle* specifies the artifact states and the events that change the artifact state, such as creation, completion and approval. The lifecycle shown in Fig. 10 is an illustrative example; different software development and management artifacts have different and often more complex lifecycles.

The artifact relationships are discussed in section 4 (“Static Structure of Software Development and Management Artifacts”). The artifact interactions are discussed in section 5 (“Dynamics of Software Development Artifacts”). The artifact types are discussed in section 3 (“Basic Features of the Object-Oriented Process Specification”).

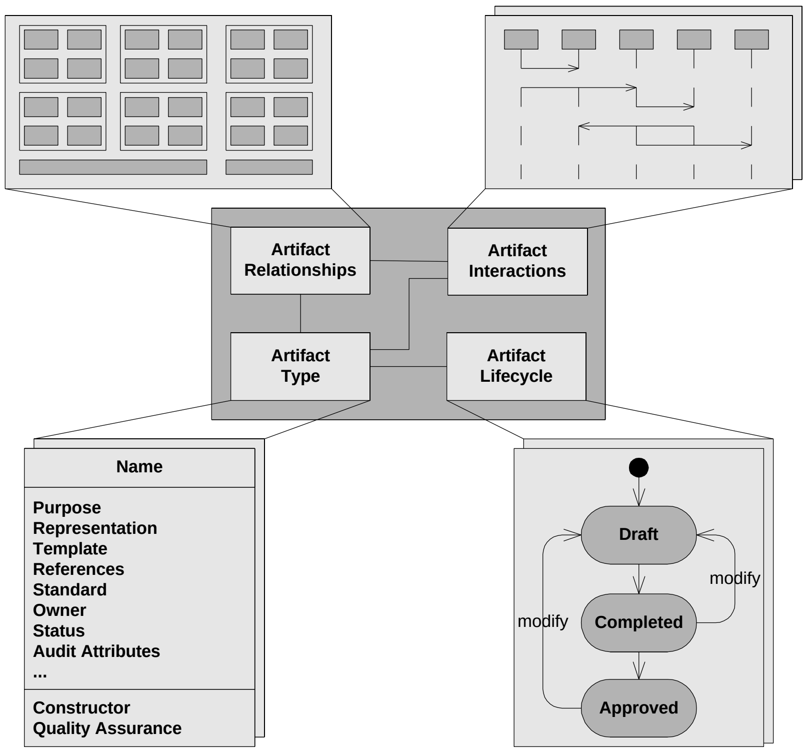

Fig. 10. Structuring the object-oriented specification of a development process

It can be noted, however, that the framework described in this article is not a so-called document-based process. Document-based processes focus on the documents delivered. The documents often have a form, which is well-defined by templates and checklists. Document-based processes have a bad reputation, because the success of a project has often been measured in terms of the documents being delivered on time. For software projects, this encourages problems, such as bureaucracy, slow delivery and requirements drift.

Many software development and management artifacts in the framework, such as class lifecycle, are too small to be useful documents delivered separately, for example, at a project milestone. For pragmatic reasons, software development and management artifacts can be combined together to make useful documents. Examples of such aggregated software development artifacts are in ref. [10].

The framework specified in this article can be characterized as information-focused, rather than document-focused. Software development and management artifacts are essentially pieces of information (see the artifact definition in section 3). The framework strictly distinguishes between the information itself (called software development or management artifact in this article) and its representation (called deliverable or document in the traditional approach). Sometimes the information is not represented physically and might exist only as a mental model. Even for such artifacts, the framework concepts can be applied.

7 Comparison to Other Design Methods and Methodological Frameworks

This section compares the original specifications of the Fusion method process, the Rational Unified Process, the OPEN process and several process definition standards, with the object-oriented specification of these processes.

The process of the *Fusion method* [3] is focused on software development artifacts, rather than on activities. For each deliverable and milestone, the Fusion method specifies how to construct the software development artifact and specifies the quality checks for each software development artifact. It is therefore quite straightforward to construct the object-oriented model for the Fusion method. The object-oriented specification of the Fusion development process (object interactions are illustrated in Fig. 9) has the following advantages over the original specification:

- The object-oriented model makes a distinction between the information itself and its representation. This allows for the alternative representation of software development artifacts in specific situations. For example, in contrast to the Fusion original representation in Backus-Naur form, lifecycles can be specified by statecharts or state tables.
- The object-oriented model allows for user-specific extensions and customization of the method. The object-oriented specification guarantees that the extensions are consistent with the original method. For example, developers might include specifications of subsystem lifecycles, class lifecycles and use cases in the original process. The object-oriented specification makes it easy to define relationships between these new artifacts and the original Fusion deliverables.
- The object-oriented model allows for consistent mapping between management artifacts and software development artifacts. In particular, the object-oriented model defines relationships between software development artifacts and the project and task, along with relationships between software development artifacts and the team model, which specifies which team roles are responsible for which deliverables.

The Rational Unified Process [12] is specified by a workflow model focused on activities. The original Rational Unified Process 5.0 [15] is complex, but it is made more manageable by viewing the process in different ways, such as by notation, workflow, documentation, artifacts and workers. It is also made more manageable by necessary configuration (adaptation) to meet specific needs. The on-line representation of the process helps significantly in dealing with the complexity of the process. The object-oriented specification of the Rational Unified Process (object interactions are illustrated in Fig. 8) has the following advantages over the original specification:

- The object-oriented model is simpler than the original one. If the set of artifacts included in the demo version of the process is used, the same information is described in 26 different documents in the demo version and in 11 documents in the object-oriented specification. The demo version was available at Rational's Web site during 1997-1998.
- The Rational Unified Process, rewritten in an object-oriented manner, provides a consistent framework: it makes inconsistencies in the original process noticeable and draws attention to missing information.
- Each artifact of the object-oriented specification has quality defined by quality-assurance methods. For comparison: the Rational Unified Process 5.0 [15] defines the quality of only about 30% of artifacts.

The Fusion process, the Rational Unified Process and their customized versions, are instances of the framework discussed in the previous section. Using the framework, these processes can be compared against each other by identifying their software development and management artifacts and specifying the order in which the artifacts are created, updated and completed, as it is shown in Figs. 8 and 9. Furthermore, the constructors, quality-assurance methods and artifact lifecycles of various processes can be compared against each other using the framework in this article.

Methodological frameworks such as OPEN and various process definition standards can be compared against the framework in this article by finding mappings between their elements. Such mappings are not analyzed in detail in this paper.

OPEN [8] is a process-focused object-oriented framework for software development methods. OPEN provides a range of activities, tasks and techniques, which can be tailored specifically to each individual organization or individual project. The OPEN process is an object-oriented framework that regards activities as objects and tasks as their methods. The execution of the objects (activities) is guarded by pre- and postconditions on their methods (tasks). The postconditions include testing requirements and deliverables. OPEN has a well-defined metamodel [4], [7] consisting of projects, activities, tasks, deliverables, techniques and sequencing rules. OPEN addresses the same problem as the framework in this article – instead of specifying a single process, it specifies the process framework and instantiates it (that is, it creates a concrete method) for specific situations. Although the aim is similar, both approaches differ in the following details.

- OPEN is a process-focused framework; this article describes a product-focused framework. OPEN objects (the first class citizens in the object-oriented process model) are *activities*. In contrast to this, objects of the framework discussed in this paper are *products.*
- Modularity and encapsulation of OPEN are at the level of *activities*. For example, OPEN allows an *activity* to be outsourced to an external organization; the contract between activities becomes a business contract between the two organizations (ref. [8], page 45). The modularity of the framework in this article is at the level of *products* (software development and management artifacts). The *product* can be obtained from an external organization, and the constructor and quality-assurance methods become a business contract between the two organizations.

- Both OPEN and the framework in this article can create specific methods by choosing the method components that meet specific demands in specific situations. A concrete method based on OPEN is created by selecting the *activities* (with tasks) necessary to perform the project; the framework in this article is instantiated by selecting the *software development and management artifacts* (pieces of information) necessary to make a final product. If the situation requires artifacts not specified in the framework, the pattern discussed in section 3 guarantees that new artifacts are added in a consistent manner.
- OPEN can be transformed into the framework in this article and the other way around. Although the artifacts in this article are smaller than OPEN deliverables, each of the OPEN deliverables can be mapped to one or more software development or management artifacts in this article, and all OPEN tasks and techniques can be expressed as the constructors of artifacts and quality-assurance methods. The OPEN activities can be mapped to collaborations between software development and management artifacts, but this mapping is only possible in certain cases. This is because the links between OPEN objects and their methods are specified in terms of probabilities (deontic certainty factors) that allocate which tasks are needed for which activities [9]. The framework in this article does not have such a probabilistic mechanism. A classifier (such as collaboration) either does or does not have its elements. Therefore, the mapping between the collaborations in this framework and OPEN activities is only possible in cases in which the values of the deontic factors change to a bimodal distribution (0 or 1).
- The mapping between OPEN and the framework in this article might be useful for the further development of OPEN. The framework in this article has a succinct structure of software development and management artifacts, and has "placeholders" for additional deliverables and tasks not included in the OPEN process specification [8]. Moreover, the framework in this article provides a pattern for extensions of the process in a consistent manner. The new improved version of the OPEN metamodel [7] indicates that OPEN process specification can easily be extended to cover software development and management artifacts from the framework in this article.

Several *standards related to process definition*, such as the Capability Maturity Model [13], ISO 9000 and the Alistair's Cockburn's VW-Staging [1] define quality criteria and the key practices that concrete development processes should meet. The object-oriented specification of a development process can be directly evaluated against the standards simply by comparing the quality-assurance methods of the software development and management artifacts with the key practices and quality criteria defined by the process definition standard. However, the process definition standards are not supported by any well-defined metamodel. The OMG Process Working Group White Paper [14] suggests a metamodel using OMG Meta Object Facility, but without regular structures and guidelines for specifications of the artifacts.

Unlike the object-oriented framework presented in this article, none of the standards mentioned describe patterns for structuring the software development and management artifacts. A development process is therefore more difficult to reuse and extend in a consistent manner. The process instantiated from the metamodel in the

OMG Process Working Group White Paper is customized off-line by means of project profiles that reflect, for example, the organizational cultures, industry domains and technology types. The idea of the profile is certainly useful. However, the object-oriented framework discussed in this paper allows for process customization at a much lower level of granularity; the processes can be customized on-the-fly to fit various *problems* being solved.

8 Summary

This paper discussed the product-focused object-oriented framework for the specification of software development processes. The software development and management artifacts are modeled as objects with constructors and quality-assurance methods, along with a number of specific attributes. The object-oriented specification of a development process is simpler and more consistent than traditional specification based on tasks and deliverables.

References

1. Cockburn, A.: Using "V-W" Staging to Clarify Spiral Development, available at: http://members.aol.com/acockburn/papers/vwstage.htm
2. Cockburn, A.: ECOOP 98 panel discussion on Software Development and Process, Brussels, Belgium, July 1998.
3. Coleman, D. Arnold, P. Bodoff, S. Dollin, C. Gilchrist, H. Hayes, F., Jeremaes ,P.: Object-Oriented Development: the Fusion method, Prentice Hall, 1994
4. Henderson-Sellers B.: A Methodological Metamodel of Process, JOOP, 11(9): 45-55, February 1999.
5. Henderson-Sellers B.: Instantiating a Process Metamodel, JOOP, 12(3): 51-57, June 1999.
6. Henderson-Sellers B.: Mellor S. J.: Tailoring Process-Focused OO Methods, JOOP, 12(4): 40-44, July/August 1999
7. Firesmith D., Henderson-Sellers B.: Improvements to the OPEN Process Metamodel, JOOP, 12(6), October 1999
8. Graham I., Henderson-Sellers B., Younessi H.: The OPEN Process Specification, Addison-Wesley, Harlow, 1997
9. Graham I.: Message in the mailing list OTUG, Subject: RE: (OTUG) Unified Process ????, 17 September 1998, 01:09 GMT.
10. Hruby, P.: Structuring Design Artifacts with UML, in: <<UML>>'98: Beyond the Notation, Bezivin J., Muller P.A. (editors), Springer Verlag LNCS 1618, 1999.
11. Hruby, P.: The Pattern for Structuring UML Based Repositories, OOPSLA'98, Vancouver, Canada, 1998.
12. Kruchten, P.: The Rational Unified Process, Addison-Wesley, 1998.
13. Paulk M. C. et al.: Capability Maturity Model for Software version 1.1, CMU/SEI-93-TR-024
14. OMG White Paper on Analysis & Design Process Engineering, Process Working Group, Analysis and Design Platform Task Force, OMG document ad/98-07-12, July 1998.
15. The Rational Unified Process 5.0, Rational Corporation, 1998.