Pattern for Structuring UML Compatible Software Project Repositories

Pavel Hruby
Navision Software a/s
Frydenlunds Allé 6
2950 Vedbaek
Denmark
E-mail: ph@navision.com
Web site: www.navision.com, click "corporate info" and "methodology"

Abstract. *Have you tried to specify your software system with UML and later experienced problems with getting an overview about the completeness and consistency of the specification and with meeting demands for usability and quality of the software design? It might help to identify a suitable structure and relevant relationships between UML software development artifacts. I will illustrate a pattern of software development artifacts that can be used for structuring project repositories containing UML diagrams. The pattern enables you to customize the size of the specification, so it matches the size of the problem and stays consistent. It allows the specification to be extended in a predictable way, if you want to specify something unusual or unexpected, such as information not covered in the UML notation guide or available literature.*

1. Motivation

Managing complexity is a major issue in software design specification. Currently, over 50 object-oriented methods have been published [3] that emphasize various aspects of software design issues and describe software development processes from various perspectives. Alistair Cockburn characterized it at ECOOP'98 panel discussion as: "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." [1].

In our approach, instead of specifying a concrete set of software development and management artifacts, we specify a framework that describes rules how to create the artifacts and relate them together. To meet the demands of specific development problems, we create specific development and management artifacts as instances of the framework. This solution significantly simplifies the description of concrete development processes because the complexity is localized, in the abstract form, in the framework.

2. Context

During the software development process, software developers identify certain information about the software product. The information can be very general, such as the vision of the product, or very concrete, such as the source code. Other examples are use cases, object collaborations and class descriptions. Software project managers specify information about management products, such as projects, project plans, organizational structures and job descriptions.

Software developers and managers want to write specifications, in which the required information about software and management products can easily be located and closely related information are linked together. The structure should also give an overview about the completeness of the specification and consistency between the artifacts.

UML (Unified Modeling Language [3]) defines a standard notation for object-oriented software systems. However, UML does not specify how to structure the information describing the software system, nor does it specify which diagrams to include in the specification or what the relationships between various diagrams are; see Fig.1.

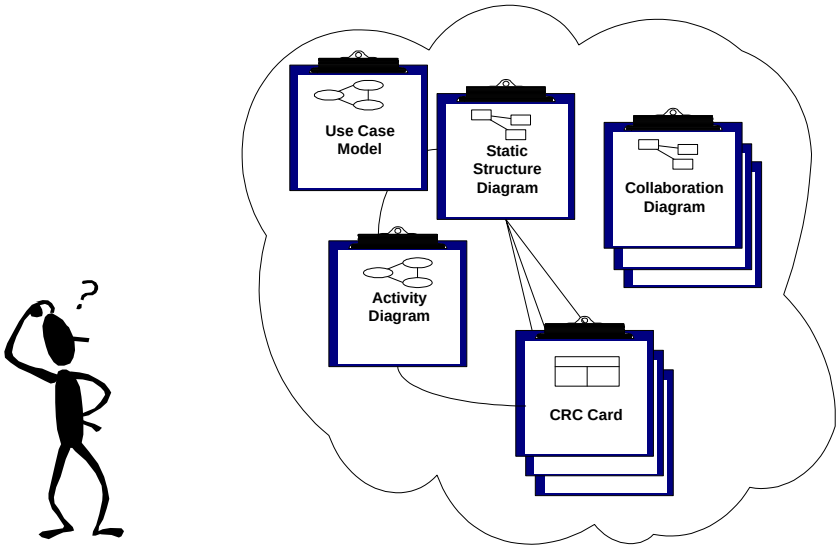

Fig.1. What to include in a good description of a software system?

3. Forces

1. A team uses a UML compatible development method for software development.
2. The team participates in software projects of various sizes, complexity and time horizons, therefore it is not possible to determine or customize a specific development method that fits all the projects.
3. The team cannot determine up front which development and management artifacts will need in each project.
4. At any time, the team wants to get an overview of the completeness of the specification and consistency between software development and management artifacts in the project repository.

4. Solution

Describe the software product from various viewpoints and describe each view[1] at various levels of granularity. At each level of granularity and in each view, consider describing the static relationships between classifiers[2], dynamic interactions between classifiers, classifier responsibilities and classifier state machines. The pattern is illustrated in Fig. 2.

In simple cases, the specification consists of a small subset of the software development and management artifacts identified in this way, but you get an overview of what information about the product is specified. The structure allows for consistent customizations. If you have to describe something unusual or unexpected, such as the things not covered by the UML or the development method of your choice, extend the specification to additional views and additional levels of granularity.

The rest of this section elaborates this idea in detail.

[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 [5], 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. Classifiers representing management artifacts, such as team and project, can be mapped to UML as stereotyped classes.

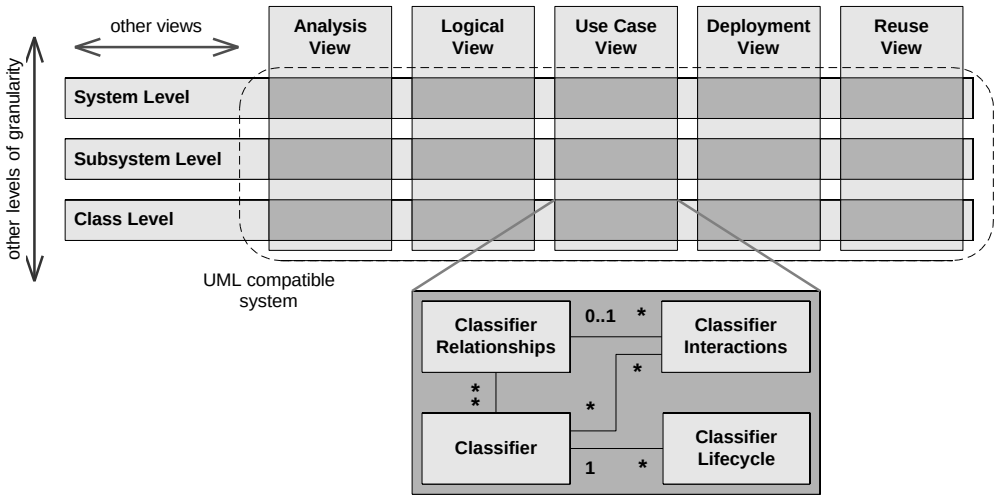

Fig.2.Viewpoints and levels of granularity.

Examples of views are the analysis view, the logical view, the use case view, the deployment view and the reuse view. The *analysis view* describes the logical structure of the product in terms of analysis objects and their responsibilities, relationships and interactions. 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. 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. Analysis objects may – but do not always – correspond to logical or physical software entities existing in the product. The *logical view* describes the logical structure of the product in terms of subsystems and classes and their responsibilities, relationships and interactions. The *use case view* identifies the system, subsystem and class use cases. The *deployment view* describes the physical structure of the system in terms of hardware devices and their responsibilities, relationships and interactions. The *reuse view* identifies reusable elements at various levels of granularity. Examples of reusable elements are reusable components, design patterns and systems of patterns.

Examples of levels of granularity are the system level, the subsystem level and the class level. 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.

In this article, I make a distinction between the term *artifact* and the term artifact *representation*. The artifact determines the information about the system, and the representation determines how the information is presented. Artifacts can be represented in a number of different ways: some development artifacts are represented in UML, some are represented by text, and some can be represented in more than one way. For example, the artifact class lifecycle can be represented by a UML statechart diagram, an activity diagram, state transition table or in Backus-Naur form. The artifact object interactions can be represented by UML sequence diagrams or by UML collaboration diagrams. Fig. 3 illustrates typical representations of software development and management artifacts.

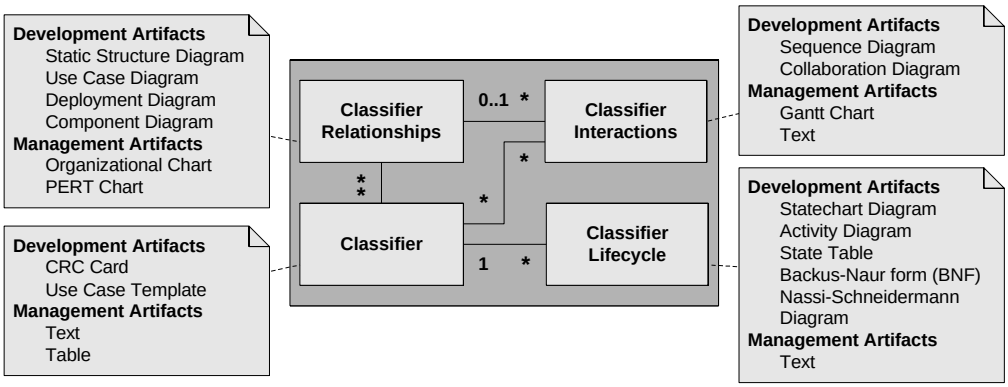

Fig. 3. Typical representation of software development and management artifacts.

The artifact *classifier relationships* specifies static relationships between classifiers. In UML, this artifact can be represented by a set of static structure diagrams (if classifiers are subsystems, classes or interfaces), a set of use case diagrams (if classifiers are use cases and actors), a set of deployment diagrams (if classifiers are nodes) and a set of component diagrams in their type form (if classifiers are components). This artifact can also be represented by an organizational chart, if the classifiers are roles or teams; or by a PERT chart, if the classifiers are tasks and projects.

The artifact *classifier interactions* specifies interactions between classifiers. In UML, this artifact can be represented by sequence diagrams or collaboration diagrams. This artifact can also be represented by a Gantt chart if the classifiers are projects; see section 5.4 for details.

The artifact *classifier* specifies classifier purpose, responsibilities and static properties of classifier interfaces, such as a list of classifier operations with preconditions and postconditions and abstract attributes that can be read and set. Classifiers can be represented by structured text, for example, in the form of a CRC card. These artifacts can also be represented by team and job descriptions if classifiers are teams and team member roles.

The artifact *classifier lifecycle* specifies classifier state machine and dynamic properties of classifier interfaces, for example, the allowable order operations and events. The classifier lifecycle can be represented by a statechart diagram, an activity diagram, a state transition table and Backus-Naur form.

5. Known Uses

This section gives examples how the pattern is used to structure the UML software development artifacts, tests, online help system, and project management artifacts, such as projects, tasks, teams and team roles.

5.1. Logical View and Use Case View

Fig. 4. specifies the software development artifacts at three levels of granularity and in the logical view and use case view. In simple cases the specification consists of only a small subset of the design artifacts identified in Fig. 4.

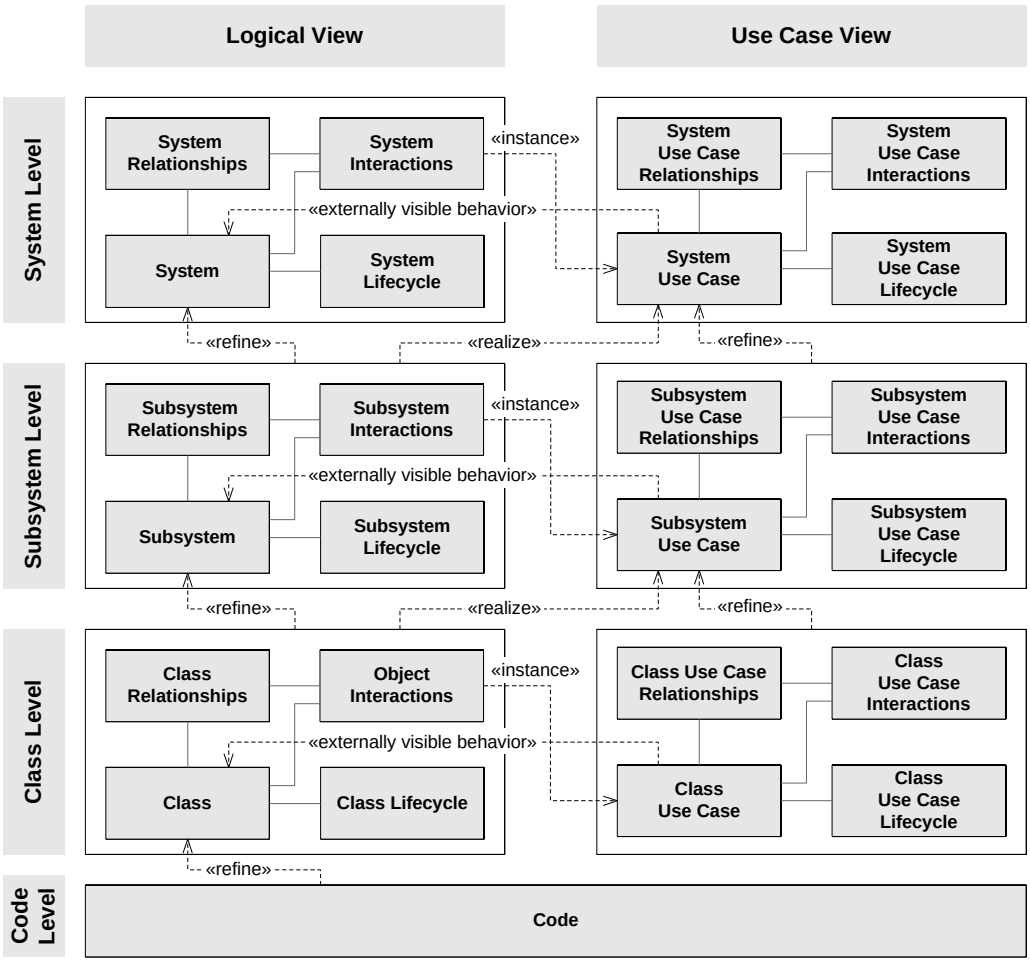

Fig.4. Software development artifacts at the system, subsystem and class levels of granularity and in the logical and use case views (after ref [2]).

As an example, the following two paragraphs outline the semantics of the software development artifacts at the system level of granularity. The artifacts at other levels of granularity can be described in a similar way.

The artifact *system relationships* specifies the context of the system – static relationships between the system and actors. The artifact *system* specifies the system responsibility and the static properties of the system interfaces, such as the interface operations with preconditions, postconditions, parameters and return values, services required from other connected systems and events that the system is capable of notifying. The *system lifecycle* specifies the abstract model of system states and dynamic properties of system interfaces, such as the allowable order of system interface operations and events. The artifact *system interactions* specifies the interactions between system and other connected systems and users. These interactions are instances of system use cases (see the dependency «instance» in Fig. 4).

The artifact *system use case* specifies the externally visible behavior of the system in terms of the use case goal, precondition, postcondition, other connected systems and users (actors) involved in the use case and their attributes accessed and modified (see the dependency «externally visible behavior» in Fig. 4). The static relationships between system use cases and actors are described in the artifact *system use case relationships*. The *system use case lifecycle* specifies the activities in the scope of the system use case. The artifact *system use case interactions* specifies scenarios in terms of typical sequences of use case instances. The system use cases can be refined into subsystem use cases at a lower level of granularity (see the dependency «refine» in Fig. 4). The design of the system use case is described at the subsystem level by subsystem relationships and interactions (see the dependency «realize» in Fig. 4).

5.2. Test View

The pattern can be used for documenting software tests with UML. Design artifacts in the test view are the *test relationships* (static relationships between tests), the *test interactions* (interactions between tests), the *test case* (a description of the test), and the *test algorithm* (a UML activity diagram describing the test algorithm). Design artifacts in the test view can be described at various levels such as the test suite level, the test level and the test script level. Artifacts at the test suite level are the *test suite* (a set of tests), the test *suite lifecycle* (the sequence of tests run within a test suite), test *suite relationships* (static relationships between test suites) and the *test suite interactions* (interactions between test suites). The dependency with the stereotype «trace» in Fig. 5 indicates that test cases can be based on use cases. The test suite relationships and interactions are not shown in the figure for simplicity.

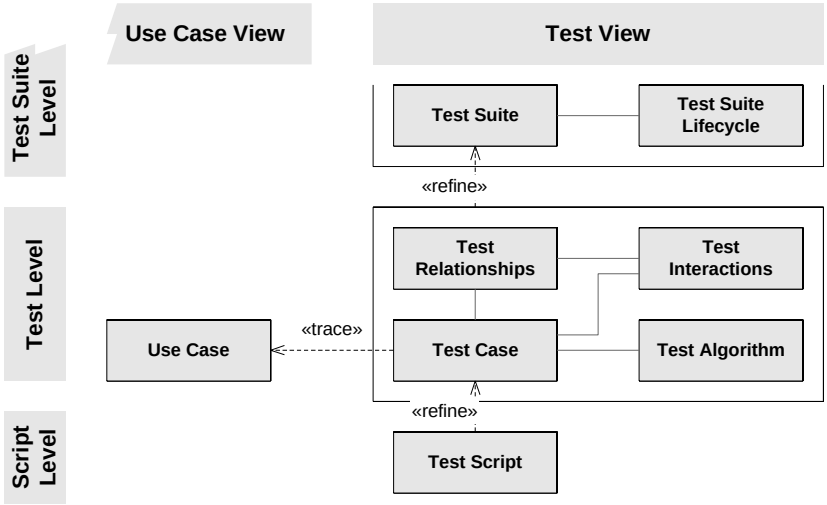

Fig.5. Software development artifacts in the test view (after ref. [2]).

5.3. Online Help View

The pattern can be used for describing the online help system. Documents (pages in online Help or Internet pages) are shown as stereotyped components in UML. Design artifacts in the online help view are the *document relationships* (static relationships between help pages), the *document interactions* (typical scenarios that arise in searching for particular information), the *document* (short descriptions of their purpose and contents) and the *document lifecycle* (if the document has behavior). The *search case relationships* specifies typical use cases how to use the user documentation and *search cases* are generalized searching scenarios. Design artifacts for online help can also be described at various levels of granularity: the book level, the document level and the text level.

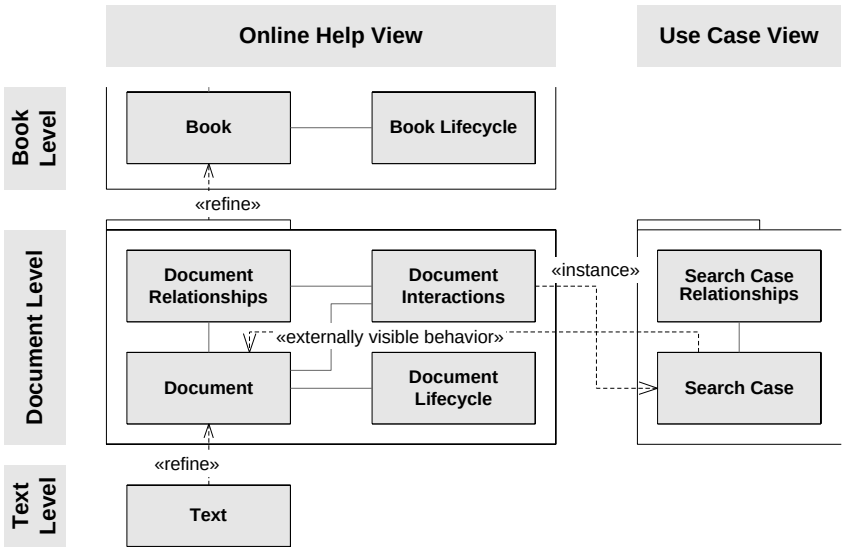

Fig. 6. Artifacts in the online help view (after ref. [2])

5.4. Team View and Project View

Fig.7 illustrates the application of the pattern for structuring management artifacts. The pattern can be used to structure two kinds of management products: 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, 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. The UML sequence diagrams can show all kinds of messages between tasks.

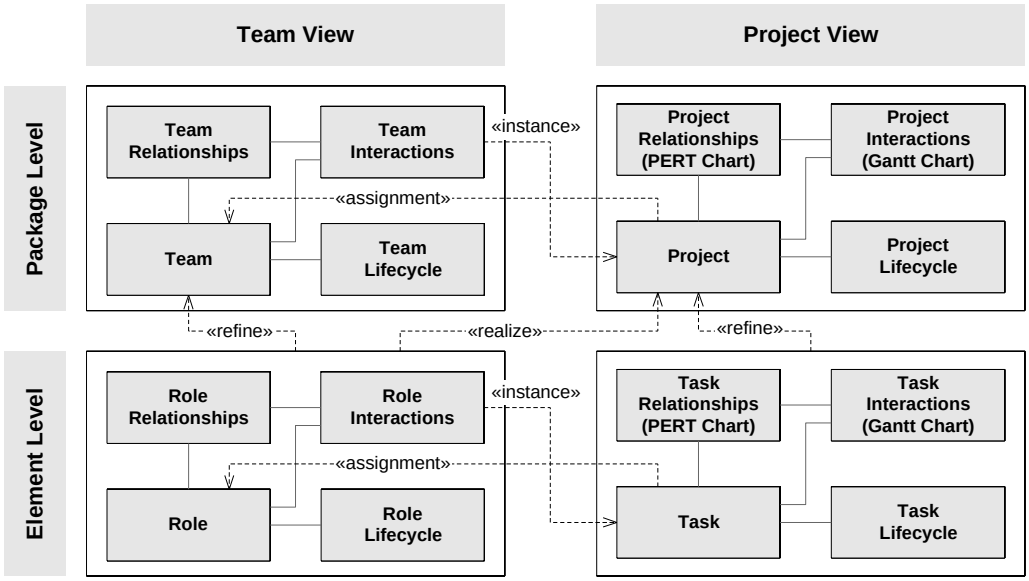

Fig. 7. Static structure of management artifact types in the team and project views.

6. Consequences

This section describes applicability and limits of the pattern.

The pattern allows for adding and removing views and levels of granularity in a consistent manner, in cases where developers and managers want to specify something unusual or unexpected. The pattern enables to customize the size of the specification, so it matches the size of the problem and stays consistent. The pattern does not force developers to create artifacts they do not need. It only gives an overview over the completeness of the specification. Fig. 8 contains examples of views not discussed in the previous section. Each of these additional views can be described at various levels of granularity and structured by using the pattern.

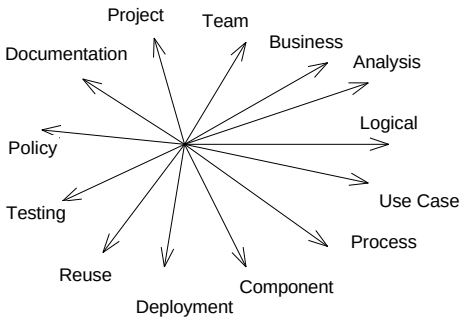

Fig. 8. Examples of views.

Small projects focus only on the limited number of views and levels of granularity. The minimal specification contains only the logical view with two development artifacts: the system and code. The artifact System specifies formally or informally the system responsibility and purpose. The artifact Code contains the source code. All other software development artifacts can be left out. As the complexity of the solution increases, developers might record important design decisions in the artifacts Object interactions, Class relationships and Component relationships.

What are the limits of the pattern? I currently understand that the pattern can describe things that we, from our point of view, want to consider as a *product*. The section 5 “known uses” illustrated examples of software development products and management products.

Every project generates a number of artifacts that we do not want to consider as a description of a software product or management product. Examples of such artifacts are a glossary of terms, minutes of meetings, reviews, comments and notes. These artifacts do not describe a product, and therefore they cannot be structured using the pattern. However, these artifacts can be related to the software development or management artifacts that are structured in the pattern. For example, the glossary is related to the artifact system and the minutes of meetings are related to the artifact project, see Fig. 9.

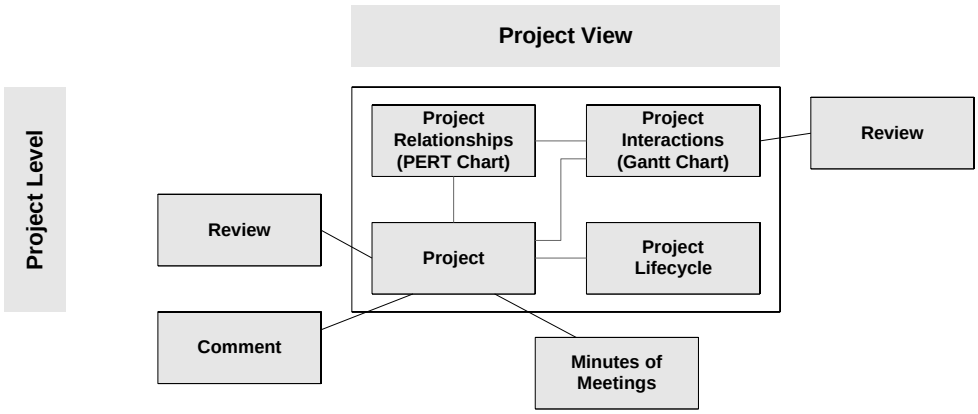

Fig. 9. Examples of artifacts that cannot be structured in the pattern.

7. Summary

Specifications of software and management products can be structured in a pattern of four mutually related software development artifacts that represent classifier relationships, interactions, responsibilities and lifecycles. The application of this pattern from different viewpoints and at different levels of granularity allows the specification to be customized in a consistent manner to cover specific software development project needs.

References

[1] Cockburn, A.: ECOOP 98 panel discussion on Software Development and Process, Brussels, Belgium, July 1998.
[2] Hruby, P.: Structuring Software Development Artifacts with UML, JOOP, 12/9, February 2000
[3] 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.
[4] UML 1.3 specification OMG document ad/99-06-08, OMG, 6 August 1999
[5] Kruchten, P.: The Rational Unified Process, Addison-Wesley, 1998.