A Pattern for Structuring UML-Based Repositories

Pavel Hrubý
Navision Software a/s, Frydenlunds Allé 6, 2950 Vedbaek, Denmark
E-mail: ph@navision.com, web site: www.navision.com (click services)

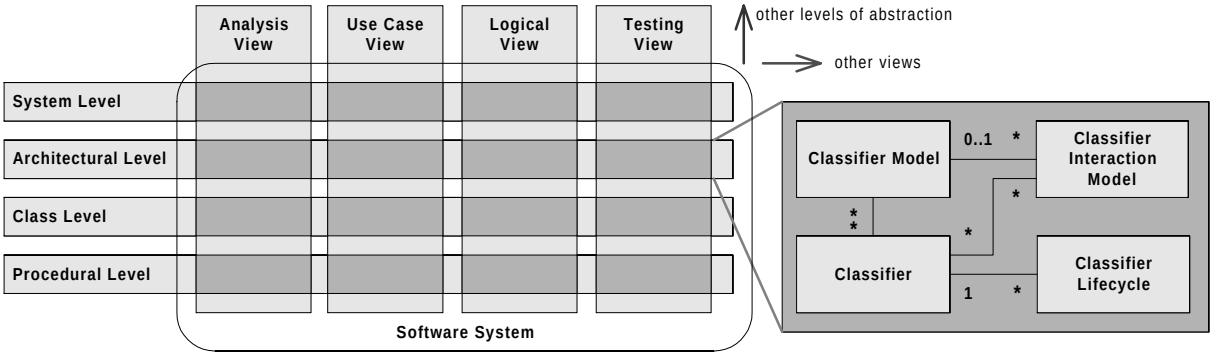
Abstract. *Have you tried to specify your software system with UML and later experienced problems with getting an overview about the completeness and consistency of the specification and with meeting demands for usability and quality of the software design? It might help to identify a suitable structure and relevant relationships between UML design artifacts. I will illustrate a pattern of design 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. After OOPSLA'98, the paper will be available for downloading at this address: http://www.navision.com (click services).*

UML Diagrams are Representations of Design Artifacts

During the development process, software architects, designers and developers identify certain information about the software product. Examples of such information are use cases, software architecture, object collaborations and class descriptions. The information can be very abstract, such as the vision of the product, or very concrete, such as the source code. In this paper, I call such pieces of information about the software product *design artifacts*.

It is useful to realize that there is a difference between a design artifact and its representation. The *design artifact* determines the information about the software system, and the *representation* determines how the information is presented. Some design artifacts are represented in UML, some are represented by text or by tables, and some are represented in a number of different ways. For example, the class lifecycle can be represented by a UML statechart diagram, an activity diagram, state transition table or in Backus-Naur form. The object interactions can be represented by UML sequence diagrams or by UML collaboration diagrams.

A useful specification of a software system is based on precisely defined design artifacts, rather than on diagrams. Design artifacts specify the software system at various levels of abstraction and in various views. At each level of abstraction and in each view, the software product can be described by four design artifacts: static relationships between classifiers, dynamic interactions between classifiers, classifier responsibilities and classifier lifecycles. Each of these artifacts can be represented by UML diagrams or by text. UML classifiers are class, interface, use case, node, subsystem and component.



At each level of abstraction and in each view, the software product can be described by four design artifacts. Each design artifact identifies specific information about the software product.

The *classifier model* specifies static relationships between classifiers. The *classifier interaction model* specifies interactions between classifiers. The *classifier* specifies classifier responsibilities, roles, and static properties of classifier interfaces (for example, a list of classifier operations with preconditions and postconditions). The *classifier lifecycle* specifies classifier state machine and dynamic properties of classifier interfaces (for example, the allowable order operations and events). All design artifacts can be represented by a UML diagram or by text. 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 [2].

The *system level* of abstraction describes the context of the system. The system level specifies responsibilities of the system being designed and responsibilities of the other systems that collaborate with it; responsibilities of physical devices and software modules outside the system; and static relationships and dynamic interactions between them and the system being designed. The *architectural level* describes subsystems, software modules and physical devices inside the system and their static relationships and dynamic interactions. The *class level* describes the detailed design of the subsystems in terms of classes and objects, their relationships and interactions, and the *procedural level* describes procedures and their algorithms.

The *logical view* describes the logical structure of the product in terms of subsystems and classes, their responsibilities, relationships and interactions. The *use case view* identifies collaborations of the system, subsystems, classes, components and nodes with actors. The *component view* describes the implementation structure of the product in terms of software modules, their responsibilities, relationships and interactions. The *deployment view* describes the physical structure of the system in terms of hardware devices, their responsibilities, relationships and interactions. The *analysis view* describes design suggestions in terms of analysis objects, their responsibilities, relationships and interactions. The software entities in the analysis view do not specify the design of the product. The purpose of the analysis view is to record preliminary or alternative solutions to design problems or to record requirements. Analysis objects may - but not always - correspond to logical or physical software entities existing in the product.

The software product can be specified at other levels of abstraction, such as the *tier level* for systems with layered architecture and the *organizational level* for business systems. The software product can be described in additional views, such as the *testing view*, the *user interface view* and the *database design view*. At each level of abstraction and in each view, the software product is specified by static relationships between classifiers, dynamic interactions between classifiers, classifier responsibilities and classifier lifecycles. The semantics of these design artifacts are outlined in [2].

Other Applications of the Pattern

The pattern can be used to structure other design artifacts, such as tests, Help documents and user interface components. It can also be used to structure project management artifacts, such as projects, tasks, teams and team roles. I will also discuss pragmatic simplifications and structuring the repository according to the system features, collaborations and refinement relationships. I will also compare the UML structure or design artifacts with the deliverable structure of the Shlaer-Mellor method, the Fusion method and the Objectory method, which do not have complete structures of design artifacts.

The Pattern and Software Development Processes

To specify a software development process, it is useful to consider design artifacts objects and evolution as collaborations between objects. Design artifacts have attributes such as name, representation, owner, project and increment identification, in addition to attributes, such as who created and modified the artifact instance and when. Artifacts have constructors that are the methods describing how to create the artifact, and quality-assurance methods, such as completeness and consistency checks. Such an object-oriented process definition can manage complexity of a development process in a better way than a description based on workflow. It is also easy to customize it *during* the process. Please contact me for more information, by sending an e-mail to ph@navision.com.

Summary

Software specifications can be structured in a pattern of four mutually related design artifacts that represent classifier relationships, interactions, responsibilities and lifecycles. The application of this pattern at different levels of abstraction and in different views to a software product allows the specification to be customized in a consistent manner to cover specific project needs.

References

[1] Hruby, P.: The Object-Oriented Model for a Development Process, OOPSLA'97, Atlanta, GA, 1997
[2] Hruby, P.: Structuring Design Deliverables with UML, <<UML>>'98, Mulhouse, France, June 3-4 1998, available at http://www.navision.com (click services)
[3] UML Semantics and Notation Guide, version 1.1, Rational, 1 September 1997