Structuring Design Deliverables with UML

Pavel Hruby

Navision Software a/s
Frydenlunds Allé 6
+45 45 65 50 00
+45 45 65 50 01
ph@navision.com

Abstract
The idea of using Unified Modeling Language (UML) appeals to people, but actually using it can be challenging. Many would like to use UML for software development, but do not know how to structure design models and what the relationships between various UML diagrams are. This paper introduces a simple structure for design deliverables that can be used for software development with UML. The structure is based on a pattern of four models describing classifier relationships, interactions, responsibilities and state machines. The pattern can be applied to different levels of abstraction and to different views on a software product. The paper also discusses practical considerations for documenting software design in the project repository as well as cases in which UML may not be the most appropriate notation to use.

Keywords
Project repository, UML, Deliverable

1. Motivation

To define the behavior of your system, some methods suggest describing scenarios, and other methods suggest creating sequence diagrams. What is the correct approach? To answer this question, we must realize that there is a difference between a design deliverable and its representation. The *deliverable* determines the information about the software product, and the *representation* determines how the information is presented. For example, a state model can be represented by a statechart diagram, an activity diagram or a state transition table. The system behavior mentioned above is determined by the system interaction model, the subsystem interaction model or the object interaction model. In UML, each of these models can be represented by a set of sequence diagrams or a set of collaboration diagrams.

Useful design documentation is based on precisely defined deliverables[1], rather than on diagrams. This paper introduces a simple structure of design deliverables that traces design information. It can easily be extended to cover all interesting information about the design of the product.

2. A Pattern of Four Deliverables

Software products can be described at various levels of abstraction and from various views. Some examples of levels of abstraction are the system level, the architectural level and the class level. Some examples of views are the logical view, the use case view and the implementation view. At each level of abstraction and in each view, the software product can be described by four artifacts: static relationships between classifiers, dynamic interactions between classifiers, classifier responsibilities and classifier state machines. Each of these artifacts can be represented either by UML diagrams or by text. The pattern is illustrated in Fig. 1.

The models in Fig. 1 represent *types* of deliverables. They define the structure and the relationships of deliverable *instances*, which contain the actual information about the software product. A model can consist of a large number of deliverable instances. For example, a class model can consist of several static structure diagrams, each of them representing small parts of a system structure; a system interaction model can consist of many interaction diagrams describing various usage scenarios. See reference [3] for more information about object-oriented deliverable models.

[1] A deliverable is a piece of information about a software product. A deliverable has a representation, properties, responsibilities, attributes, methods and relationships to other deliverables. See also reference [3].

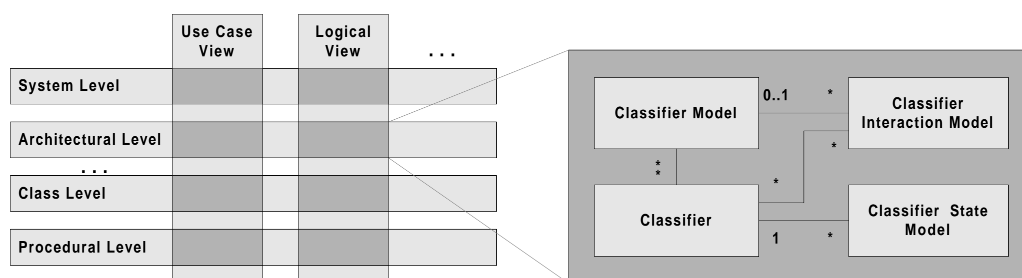

Fig. 1. At each level of abstraction and in each view, the software product can be described by four deliverables. UML classifiers are class, interface, use case, node, subsystem and component.

The *classifier model* specifies static relationships between classifiers. The classifier model 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). The classifier model can also be represented by tables (see section 7 for details).

The *classifier interaction model* specifies interactions between classifiers. The classifier interaction model can be represented by interaction diagrams: sequence diagrams or collaboration diagrams. 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 section 6 of this paper.

The deliverable called *classifier* specifies classifier responsibilities, roles, and static properties of classifier interfaces (for example, a list of classifier operations with preconditions and postconditions). Classifiers can be represented by structured text, for example, in the form of a CRC card.

The *classifier state model* specifies classifier state machine and dynamic properties of classifier interfaces (for example, the allowable order operations and events). The classifier state model can be represented by a statechart diagram, an activity diagram, a state transition table and Backus-Naur form (see reference [7]).

An instance of the *classifier model* can be linked to several instances of the *classifier interaction model*. All of these instances are linked to instances of the *classifier*. An instance of the *classifier* is linked to an instance of the *classifier state model*.

3. Applying the Pattern

Figs. 2 and 3 show the pattern applied in use case, logical, component and deployment views, because UML is intended preferably to be used in these areas. In Figs. 1, 2 and 3, the product is described at four levels of abstraction: the system, architectural, class and procedural levels. Section 5 discusses application of the pattern at several other levels of abstraction and views on the software product.

The *system level* describes the context of the system. The system level specifies responsibilities of the system being designed and responsibilities of the 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 a*rchitectural level* describes subsystems, software modules and physical devices inside the system and their static relationships and dynamic interactions. The c*lass level* describes classes and objects, their relationships and interactions, and the *procedure level* describes procedures and their algorithms. Many large systems have additional abstraction levels, which, for the sake of simplicity, are not shown in Figs. 1, 2 and 3. For example, systems with layered architecture have an extra *tier level* between the system level and the architectural level. The *tier* level specifies system layers, their relationships and interactions. In a layered system each layer contains subsystems and components, which are specified at the architectural (subsystem) level. Some development processes also require one or more *levels of analysis models* for identifying requirements.

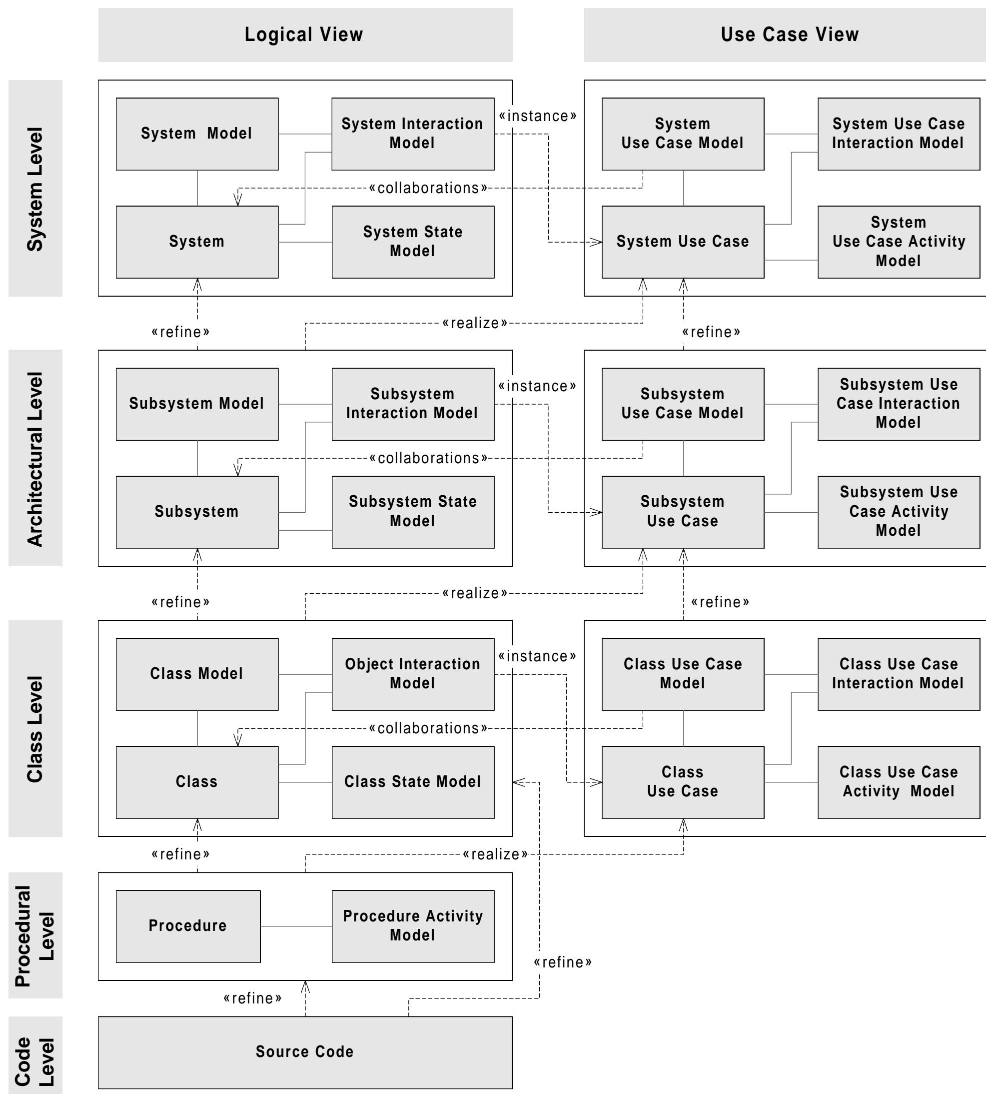

Fig. 2 Deliverables describing the software product in use case and logical views.

As an example, the text in the following paragraphs describes deliverables and their relationships at the architectural level. At all other levels of abstraction, the pattern is applied in a very similar way. The only exception is the procedural level, which does not contain the procedure model (relationships between procedures) or the procedure interaction model (interactions between procedures). The reason for the absence of models is the principle of object-oriented design, in which the class model and the object interaction model substitute procedure relationships and procedure interactions respectively.

The s*ubsystem model, subsystem component model, and subsystem node model* specify static relationships between subsystems, software modules and physical devices inside the system.

The *subsystem use case model* describes use cases with subsystem scope and their relationships to collaborating subsystems. The subsystem use case model specifies how the subsystem, its software modules and physical devices collaborate[2] with other subsystems or external actors. The dependency with the stereotype «collaborations» in Figs. 2 and 3 indicates that the use case model specifies collaborations of subsystem, component and node.

[2] UML 1.1 does not have a symbol for collaboration. Therefore, in this article I assume that collaborations are specified by use cases.

The *subsystem interaction model, subsystem component interaction model and subsystem node interaction model* describe interactions between subsystems, interactions between software modules and interactions between nodes inside the system. The dependency with the stereotype «instance» in Figs. 2 and 3 indicates that interactions specified in these models are instances of subsystem use cases.

The deliverables *subsystem*, *component* and *node* specify responsibilities of subsystems, software modules and physical devices inside the system. These deliverables also specify their own roles and static properties of their interfaces (for example, a list of operations and events). A dependency with the stereotype «refine» indicates that the deliverables class model, object interaction model, class state model and class, represent detailed design of the subsystem.

The s*ubsystem state model, subsystem component state model and subsystem node state model* specify behavior of subsystems, software modules and physical devices inside the system. In particular, they specify dynamic properties of their interfaces, for example, the allowable order of their operations and events.

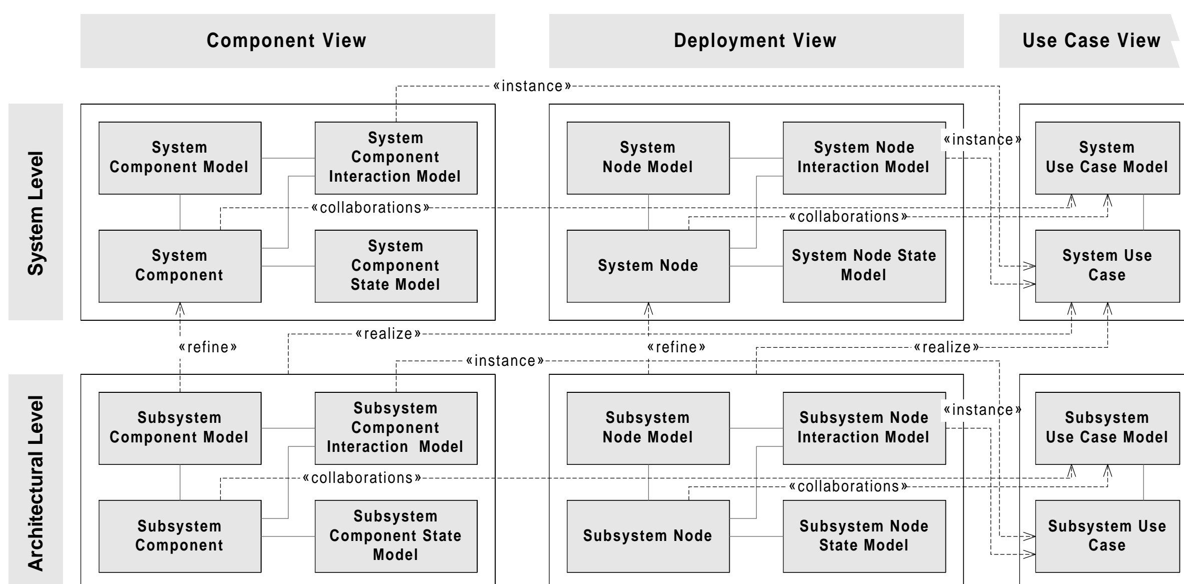

Fig.3. Deliverables describing the software product in component and deployment views.

The s*ubsystem use case* describes responsibilities of a use case with subsystem scope. This deliverable specifies static properties of the use case, for example, use case goal, pre- and postconditions, list of subsystem operations that are called within this use case, or a list of objects and attributes that are accessed or modified by the use case. The dependency with the stereotype «instance» indicates that interaction models at the subsystem level represent instances of the subsystem use case. The dependency with the stereotype «realize» indicates that a cluster of four deliverables at the class level represents realization of the subsystem use case.

The s*ubsystem use case activity model* specifies behavior of the subsystem within the scope of the use case. The subsystem use case activity model specifies subsystem state transitions and the allowable order of subsystem operations and events, which are relevant for this use case. The use case activity model can divide potentially complex state models of the subsystem into several state models of subsystem use cases, which can be simpler. The scope of the use case activity model is limited to a particular use case, in contrast to the subsystem state model, which completely describes the behavior of the entire subsystem. Another difference is that the subsystem state model is associated with the subsystem, while the use case activity model is associated with the use case.

Some methodologists suggest that activities in the system use case activity model can be associated with subsystem use cases. This suggestion does not entirely reflect the spirit of UML. However, if we accept the suggestion, then the system use case activity model can specify the allowable order of subsystem use cases. In other words, it can specify a scenario consisting of subsystem use cases.

The *subsystem use case interaction model* specifies typical sequences of use case instances. In contrast to the subsystem, component and node interaction models, where a scenario is described as a sequence of messages, the use case interaction model describes the scenario as a sequence of use cases. This model is the only UML deliverable that can describe a scenario consisting of other scenarios. This deliverable also differs from the use case activity model. The use case activity model *completely* describes the subsystem behavior within the use case, and it is related to the subsystem use case. The use case interaction model describes only *typical scenarios*, consisting of subsystem use cases, and it is related to the subsystem use case model. There are more details about the use case interaction model in section 6.2.

The UML system of diagrams is not orthogonal. In other words, the same information can be specified in two or more different UML diagrams. For example, both the static structure diagram and the object collaboration diagram specify relationships between objects, and both statecharts and interaction diagrams specify messages between objects. Because the same information can be specified in several places, models either have to be checked for consistency, or users must produce only a certain subset of the deliverables identified in Figs. 2 and 3. In the latter case, it is quite important to specify clearly which deliverables are produced and which aspects of the system are documented. It is particularly important in the case of simple software products, which are often described sufficiently using only several of the deliverables discussed in this section.

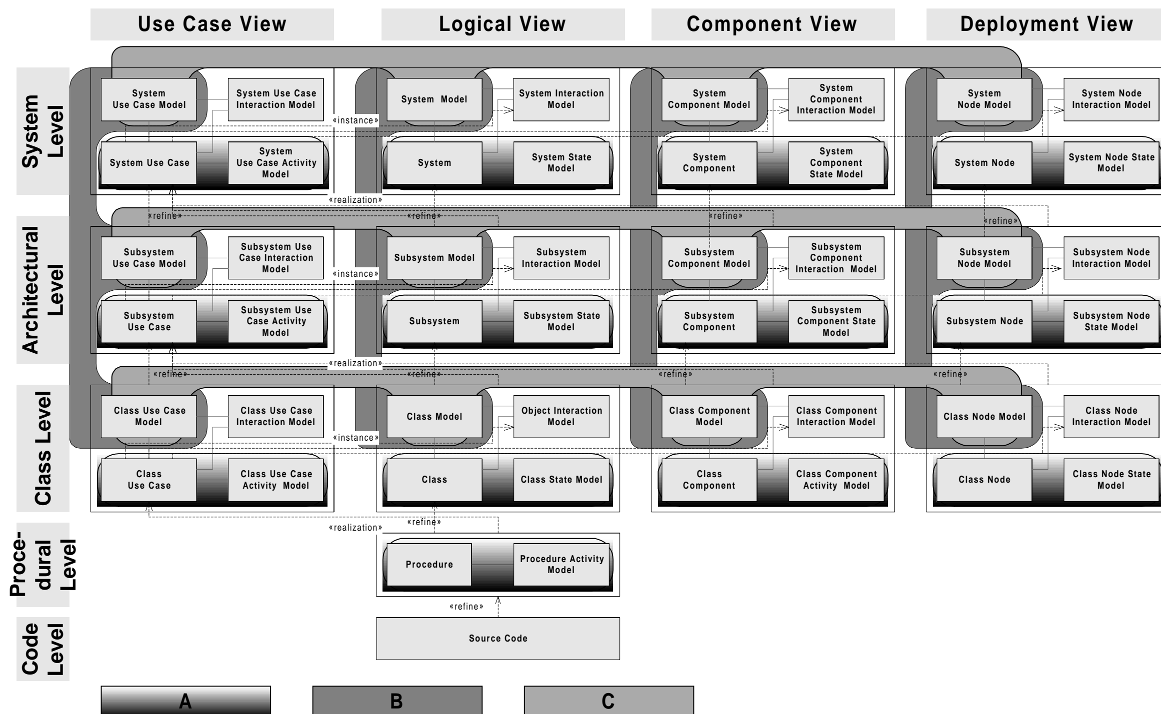

Fig.4. Several ways how to simplify the structure by joining closely related deliverables.

The system of deliverables discussed in this section can be simplified in various ways. Typically, instances of deliverables are separate documents. However, there might be pragmatic reasons for creating documents containing several closely related deliverables. For instance, classifier responsibilities and state machines are always related together and can be joined into one document (Fill pattern A in Fig.4). It is also possible to join system, subsystem and class use case models to one use case diagram (Fill pattern B in Fig.4), providing that use case levels and relationships between use cases and other deliverables are clearly distinguished. Similarly, component and node models at all levels can be joined into one implementation diagram document, providing that levels of components and nodes are distinguished. It might also be reasonable to create one static structure model within each level and show static relationships between use cases, actors, subsystems, classes, components and nodes in one diagram (Fill pattern C in Fig.4), although the *UML Notation Guide* does not mention such a combined static structure diagram.

4. Structuring Design Deliverables

In well-structured design documentation, the required information about software products can be easily located and closely related information is linked together. It also gives an overview about the completeness of the documentation and consistency between deliverables. This section proposes three rules that help to structure project deliverables in an efficient way. The rules are based on the relationships between the deliverables identified in sections 2 and 3.

The first rule is that relationships *among the four deliverables in the pattern*, shown in Fig. 1 are the closest relationships between deliverable instances. For example, an instance of the class model is linked to several instances of the object interaction model. All of them are linked to several instances of the class, and each instance of the class is linked to an instance of the class state model. Structuring deliverables in this way provides an overview of the product within the scope of the level of abstraction and the view. However, this rule is not sufficient in cases in which some of the models consist of large numbers of deliverable instances. In such cases, the following two rules, which describe relationships crossing levels of abstraction and views, must be applied.

The second rule structures deliverables according to *collaborations*. These relationships are shown in Fig. 2 and Fig. 3 as dependencies with the stereotypes «instance», «realize» and «collaborations». In Fig. 5, these dependencies are refined to associations because associations are more descriptive than dependencies. For example, the *system use case model* contains a package of use cases. This package is linked to the deliverable *system*, which specifies the system responsibility in the scope of this use case package. Responsibility of each use case in the package is specified in the *use case*. Instances of these use cases are shown in the *system interaction model,* and their realizations are specified in the logical, implementation and deployment views as a cluster of four deliverables at the architectural level. Structuring deliverables according to collaborations (their relationships to a use case) is useful for understanding the system functionality in a particular context.

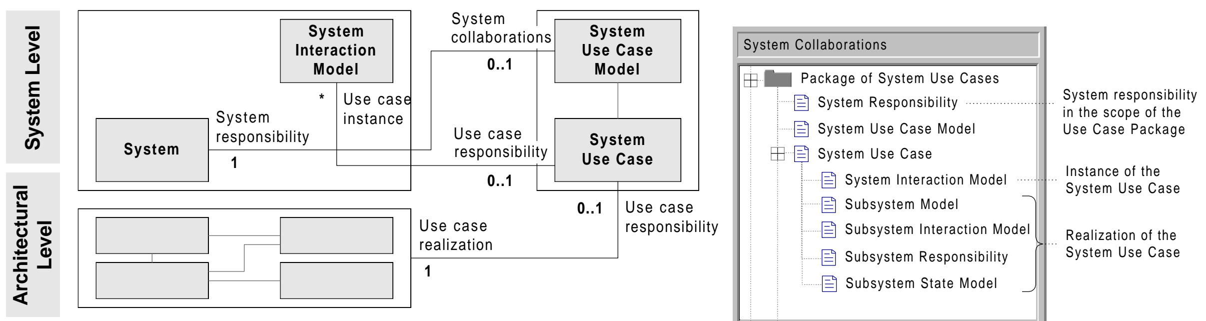

Fig. 5. Structuring deliverables according to collaborations specified in the use case model. Associations between deliverables are on the left and an example of their projection is on the right.

Structuring deliverables according to collaborations can make it difficult to see the overall structure and functionality of the system, component or class. Therefore, the third rule structures design deliverables according to their *refinement between levels of abstraction.* These relationships are shown in Fig. 2 and Fig. 3 as dependencies with a stereotype «refine», and in Fig. 6 these dependencies are refined to associations between deliverables. For example, system responsibilities and system interfaces are defined in the deliverable *system*. The s*ubsystem model* specifies the static structure of the system, and the *subsystem interaction model* specifies the design of each operation in the system interface in terms of subsystem interactions. The dependency «conform» indicates that the operation design has to match the dynamic properties of the system interface specified in the system state model.

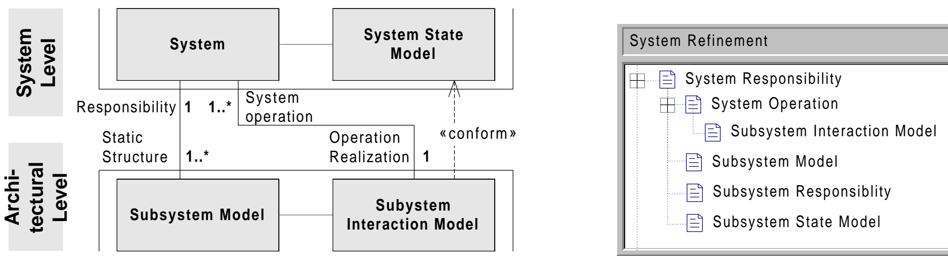

Fig. 6. Structuring deliverables according to their refinement between levels of abstraction. Associations between deliverables are on the left and an example of their projection is on the right. Components and nodes can be structured in the same way.

All three rules, *relationships within the view and level of abstraction, collaborations* and *refinement between levels of abstraction* can be combined if a project repository uses these rules as indexes. If project documentation is saved in a version control system with a single index, or, if the documentation is paper based, then a designer must choose one of these rules. Typically, it is useful to structure high-level documents according to the collaborations and low-level documents according to their refinements.

5. Other Applications of the Pattern

The pattern can be applied in different areas to describe various aspects of the system. This section discusses application of the pattern at the domain level, in analysis models, in designing software tests and in designing user documentation.

5.1. Domain Level Models and Analysis Models

The *domain level* describes the problem domain in terms of domain objects and their interactions. The domain level contains the domain model (relationships between domain objects), the domain object interaction model (interactions between domain objects) and responsibilities and state machines of domain objects. Domain use cases are use cases with "organization" scope (see reference [1]). Models at the domain level are usually refined into models at the system or subsystem level.

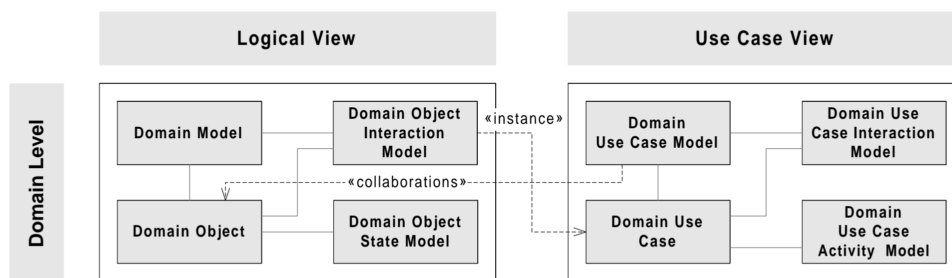

Fig. 7. Deliverables at the domain level.

The same idea can be used to produce *analysis models* at the system, the subsystem and possibly even the class levels (see reference [2]). Analysis models do not specify the design of the product, their main purpose is to identify the requirements for the product. Analysis models contain analysis objects, their interactions, responsibilities and state machines. Analysis models are typically refined into the models in the logical view, shown in Fig. 2. However, they can also be refined into the implementation models shown in Fig. 3.

5.2. Testing

The pattern can be used for designing tests. Deliverables in the test view are the test model (static relationships between tests), the test interaction model (interactions between tests), the test case (description of the test), and the test algorithm (test activity model describing the test algorithm). Test deliverables can be described at various levels such as the test suite level, the test level and the test script level. Deliverables at the test suite level are the test suite

(a set of tests), the test suite activity model (the sequence of tests run within a test suite), test suite model (static relationships between test suites) and the test suite interaction model (interactions between test suites). The dependency with the stereotype «trace» in Fig. 8 indicates that test cases can be based on use cases.

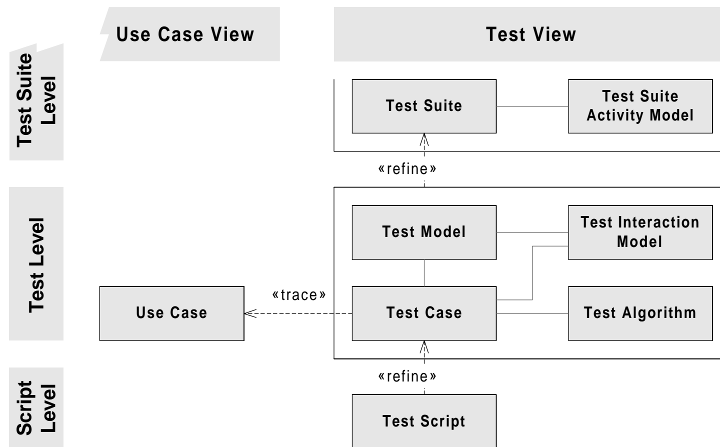

Fig.8. Deliverables for test design.

5.3. User Documentation

The pattern can be used for designing online user documentation. Documents (pages in online Help or Internet pages) are shown as stereotyped components in UML. Deliverables for designing user documentation are the document model (static relationships between documents), the document interaction model (typical scenarios that arise in searching for particular information), responsibilities of documents (short descriptions of their purpose and contents) and document state model (if the document has behavior). Deliverables for user documentation can also be described at various levels: the book level, the document level and the text level.

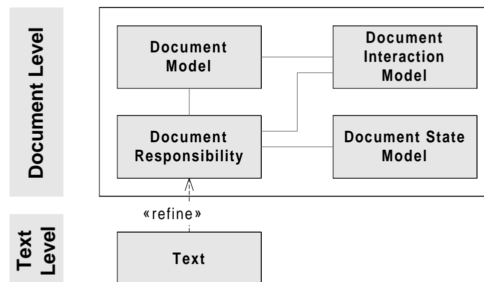

Fig. 9. Design of user documentation.

5.4 User Interface

The pattern can be used for designing user interface. Screens (windows) can be shown as stereotyped classes in UML. Deliverables for designing user interface are the screen model (static relationships between screens), the screen interaction model (typical sequences of activation of screens), responsibilities of screens (with their drawings, for example), and screen state model (if the screen has behavior). The dependency with the stereotype «instance» in Fig. 10 indicates that screen interactions are instances of use cases.

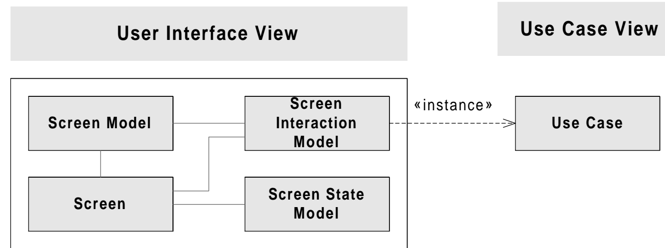

Fig.10. Deliverables for design of user interface.

6. Less Common UML Diagrams

Fig. 2 and Fig. 3 show several models that can be represented by UML, but diagrams of them are not explicitly mentioned in the *UML Notation Guide* (see reference [5]). They are the use case interaction model, the subsystem interaction model, the node interaction model and the component interaction model. These models can be represented by sequence or collaboration diagrams in which classifier roles are use case, subsystem, node and component roles.

In UML 1.1, classifier roles in sequence and collaboration diagrams are shown as objects. This might lead to confusion in cases of interactions between classifiers of different kinds. For example, symbols on the collaboration diagram, which represents interactions between the object, subsystem and component, are all shown as objects. Sequence and collaboration diagrams would be easier to understand if an object symbol representing the classifier role was replaced by the symbol of an actual classifier, as shown in Figs. 11 and 12.

6.1 Interaction Diagrams for Subsystem, Component and Node Interactions

Interaction diagrams for subsystem, component and node interactions are sequence and collaboration diagrams in which classifiers are subsystem, component and node. These diagrams represent interactions between subsystem, component and node instances, without it being necessary to specify actual objects that send or receive messages. Fig. 11 shows a collaboration diagram representing interactions between objects and subsystems.

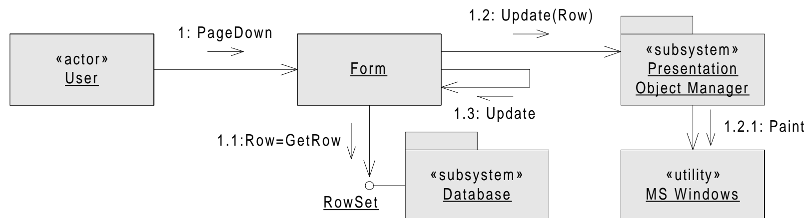

Fig. 11. Collaboration diagram representing subsystem interaction model. The notation is modified UML. In UML 1.1, all symbols are replaced by rectangles.

6.2 Diagrams for Use Case Interactions

Use case interaction diagrams are sequence and collaboration diagrams in which classifier roles are use case roles. This type of diagram can represent scenarios consisting of sequences of use cases. An actor can use a system in a way that initiates use cases in a particular order. Such a scenario – a sequence of use cases – can provide useful information about the system, and it can be shown in use case interaction diagrams.

Please note that use cases in UML can interact only with actors and not with each other. Also, they are always initiated by a signal from the actor. Therefore, the label *invoke* in Fig. 12 means that an *actor* can invoke a use case while executing another use case. Invocations on the diagram map to signals from an actor to a use case and to static relationships between use cases: generalizations «uses» and «extends», dependencies «invokes» and «precedes», or constraints {invokes} and {precedes}.

Please note that the complete behavior (not just scenarios) of a specific use case can be described in activity or state diagrams in which states or action states map to subordinate use cases.

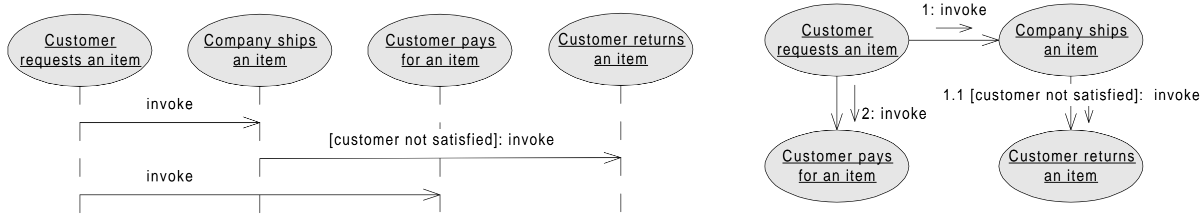

Fig. 12. Example of sequence and collaboration diagram representing use case interaction model. The notation is modified UML. In UML 1.1, ellipses are replaced by rectangles.

7. Alternative Presentation Options

Design deliverables do not necessarily have to be described by UML. Practical alternatives to UML are Backus-Naur form (BNF), tables and text. The choice of the representation depends on the problem being described, as well as other circumstances such as who the intended reader is.

Backus-Naur form (BNF) (see references [2] and [7]) represents scenarios with one or two participants or a valid order of operations of one classifier. Therefore, BNF is convenient for specifying interfaces. Fig. 11 shows an interface with five operations, where the operation `Create` must be called first, and the operations `Read`, `Write` and `Print` can then be called in arbitrary order. The operation `Delete` must be called last. A BNF expression of this scenario is as follows: `Create(); (Read() | Write() | Print() )*; Delete().` In simple cases, BNF expressions can be placed directly into the operation compartment of the class, as is shown in Fig. 13.

IPDDriver
{abstract}

«BNF»
create() (
read() |
write() |
print())*
delete()

Fig. 13. Allowable order of interface operations can be specified in Backus-Naur form.

Tables can describe relationships between classifiers, states or other entities that can have mutual relationships. Although a diagram is a more user-friendly representation, a table is a good development tool and ensures that *all* relationships between entities have been considered. For example, a table describing relationships between classes has class names in rows and columns and relationships between classes are specified in the table fields. State transition tables are a presentation alternative to statechart diagrams or activity diagrams. Rows of state transition table represent states, columns represent events and table fields contain conditions and actions of state transitions.

Structured or free text can be used to describe classifier responsibilities. Text can be structured in a way that is similar to the way a CRC card is structured.

8. Systems of Deliverables of Other Development Processes

Depending on which aspects of software design they focus on, different UML-based development processes use only certain subsets of the deliverables identified in section 3. This section compares the design deliverables of three major development processes: the Objectory method, the Shlaer-Mellor method and the Fusion method.

Although the *Objectory* method (see reference [5]) specifies deliverables with a wide scope, from a product vision to release notes and training materials, it is quite superficial in its specification of the structure of deliverables containing information about the design of the software product. The deliverables are structured on use case, logical, deployment, implementation and process views, and tier, architectural, and class levels. Deployment and implementation views contain only component and node models and component responsibilities. All interaction models are considered as a specific view called *process view*. The method produces only use cases at the system

level; the method does not produce any state models with the exception of the use case activity model and the class state model. The deliverables are structured according to their relationships to use cases (in other words, according to their collaborations with external actors).

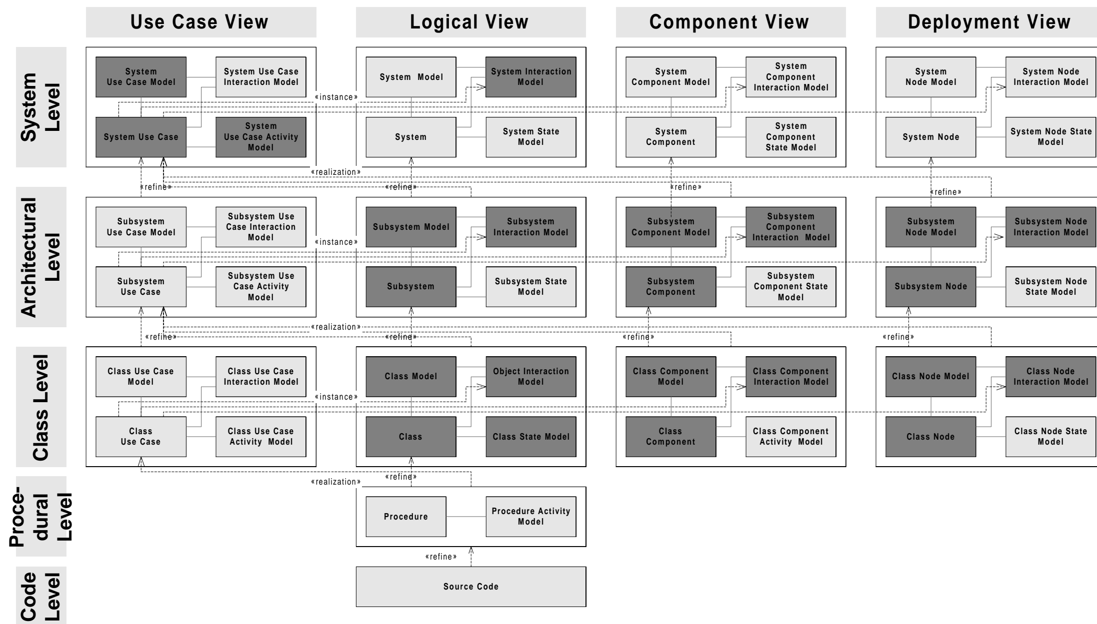

Fig. 14. Deliverables of Objectory method are shown in gray color.

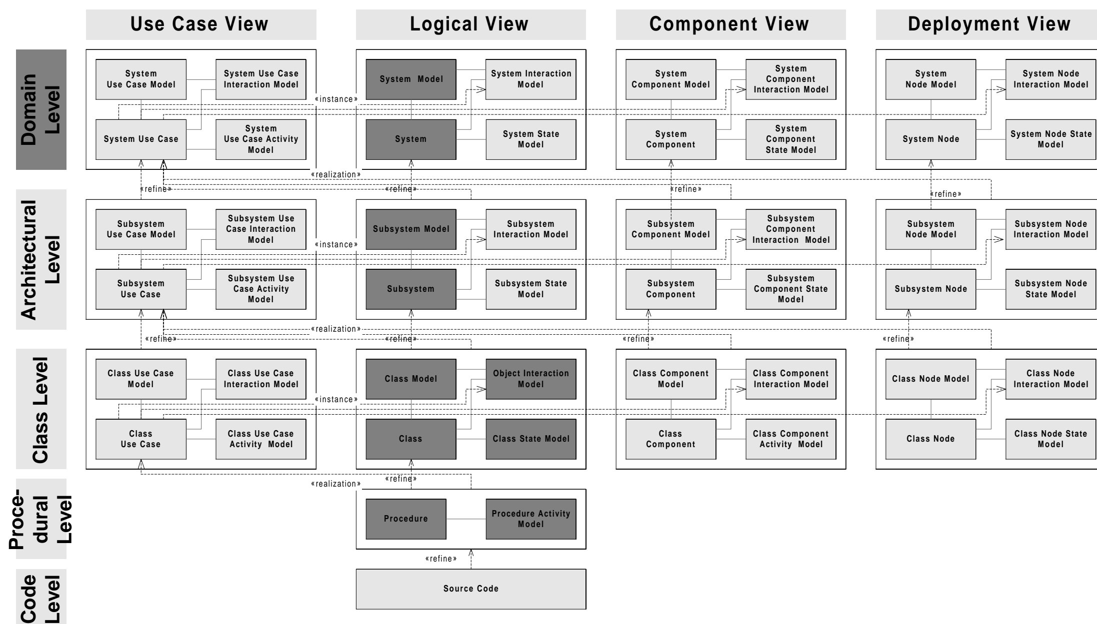

Fig. 15. Deliverables of Shlaer-Mellor method are shown in gray color.

The *Shlaer-Mellor method* (see reference [6]) has one of the best systems of deliverables. Unlike the system in Figs. 2 and 3, the deliverable system of the Shlaer-Mellor method is orthogonal, which means that one fact about the product is stated only in one place. Analysis in the Shlaer-Mellor method (hereafter SM) is focused on the logical view, and therefore the method does not produce any deliverables in use case, component and implementation views.

The Shlaer-Mellor method does not produce any deliverables at the system level. The method recognizes an extra domain level (see section 5) with the domain model (called *domain chart* in SM). At the subsystem level, the method produces the subsystem model (*subsystem relationship model* and *subsystem access model* in SM), the subsystem interaction model (*subsystem communication model* in SM) and the subsystem (*subsystem description* in SM). At the class level the Shlaer-Mellor method produces the class model (*object information model* and *object access model* in SM), the object interaction model (*object communication model* and *thread of control chart* in SM), the class (*object description* in SM) and the class state model (*state transition diagram* and *class structure chart* in SM). At the procedure level, Shlaer-Mellor produces the procedure (*action specification* in SM) and the procedure algorithm (*action data flow diagram* in SM). Please note that the procedure (*action specification*) is related directly to the state in SM and not first to the class and then to the state as it is in Fig. 2.

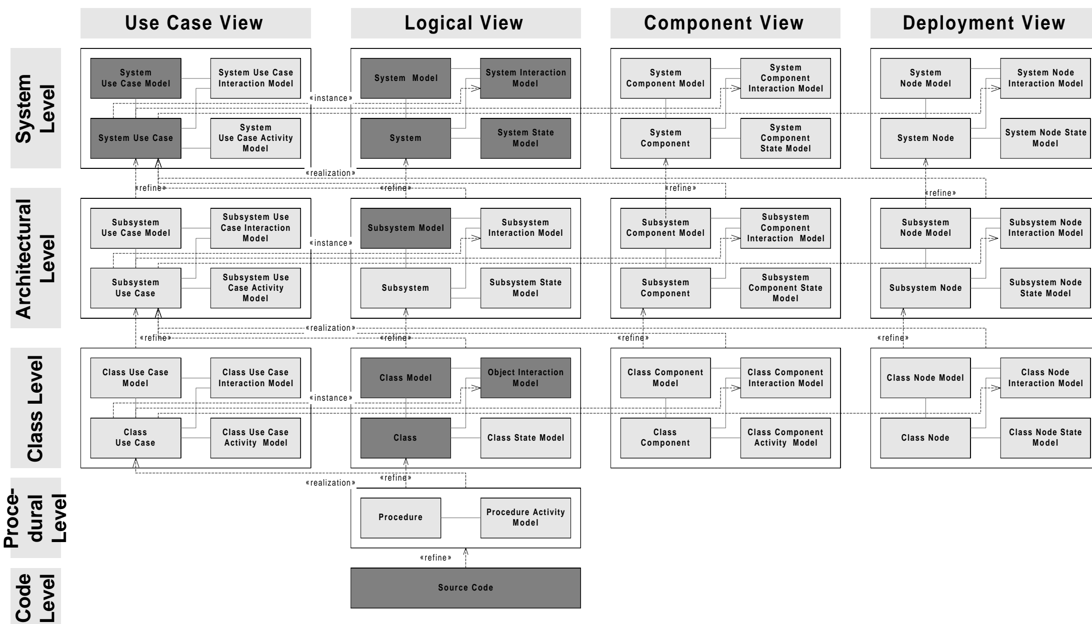

Fig. 16. Deliverables of Fusion method are shown in gray color.

The *Fusion* method (see reference [2]) is a method with a succinct and consistent system of deliverables that is also orthogonal and significantly simpler than Shlaer-Mellor. Fusion focuses on deliverables in the logical view at system, subsystem and class levels. At the system level, Fusion delivers the system model (*object model* in Fusion), the system interaction model (*scenario* in Fusion), the system (*operation model* in Fusion) and the system state model (*lifecycle model* in Fusion). At the subsystem level, Fusion delivers only the subsystem model (*system object model* in Fusion). At the class level, Fusion delivers the class model (*visibility graphs* and *inheritance graphs*), the object interaction model (*object interaction graphs*) and the class (*class descriptions* in Fusion). Fusion does not produce any state models except of the system state model (*lifecycle model* in Fusion). New Fusion Engineering process (also known as Team Fusion) produces also use case model and use cases. Deliverables are structured according to the refinement between levels of abstraction.

9. Summary

This paper introduced a pattern of four mutually related design deliverables that represent classifier relationships, interactions, responsibilities and state machines. The pattern was applied for different levels of abstraction and for different views on a software product. Application of the pattern helped to identify new interaction diagrams not documented in the *UML Notation Guide*. They are the use case interaction diagram, the subsystem interaction diagram, the node interaction diagram and the component interaction diagram. The paper outlined purpose, relationships and representation of deliverables often used to document software design. The paper also discussed three rules of structuring project deliverables based on: (1) *relationships among the four deliverables in the pattern* (2) *collaborations* and (3) *refinement between levels of abstraction*. The pattern can be easily extended to document

various aspects of software design. The paper discussed four of these aspects: domain and analysis models, documentation of test design, design of user interface and design of online user documentation.

References

[1] Cockburn, A.: Using Goal-Based Use Cases, Journal of Object Oriented Programming, November 1997, also available at http://members.aol.com/acockburn/papers/usecases.htm
[2] Coleman, D. et al.: Object-Oriented Development: The Fusion Method, Prentice Hall, Inc. 1994
[3] Hruby, P.: The Object-Oriented Model for a Development Process, OOPSLA97, also available at http://www.navision.com/services/default.asp
[4] Rational Objectory Process 4.1, demo version, available at http://www.rational.com
[5] UML Notation Guide, version 1.1, Rational, 1 September 1997, also at http://www.rational.com/uml
[6] Shlaer, S., Mellor, S. J.: Object Lifecycles: Modeling the World in States, Prentice Hall, Inc. 1992
[7] Thibault, E.: What is BNF Notation? Available at http://cuiwww.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html