NAVISION software

THE PATTERN FOR STRUCTURING UML-BASED REPOSITORIES

Pavel Hruby
Navision Software a/s
E-mail: ph@navision.com
Internet: www.navision.com

Pavel Hruby, Ph.D. works as a methodologist at Navision Software.

Navision Software is a strategic provider of efficient enterprise business solutions. Navision Software was founded in Denmark in 1984 and is 100% privately owned. More than 32,000 Navision solutions have been installed worldwide in 75 countries.

Address:	Frydenlunds Allé 6 2950 Vedbæk, Denmark
Internet:	http://www.navision.com. Click *services* to access the methodology area.

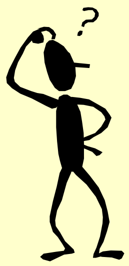
You can reach me here near the poster from Tuesday to Thursday at the end the lunch breaks (about 1 p.m.). After OOPSLA'98, please send an e-mail to ph@navision.com.

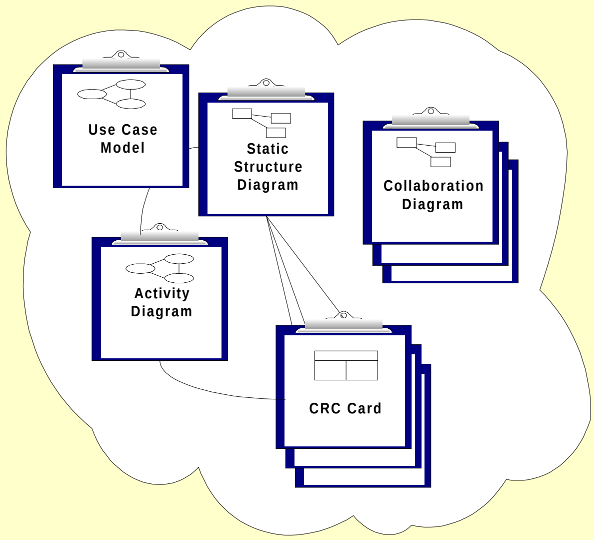
If you need help with specifying your system with UML, or if you want to hear more about my experience with UML, please feel free to contact me.

If you have comments or knowledge you are willing to share, I am interested in hearing your opinion.

NAVISION software

What Makes A Good Description of a Software System?





UML (Unified Modeling Language) 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.

In well-structured design specification, the required information about a software product should easily be located and closely related information should be linked together. The structure should also give an overview about the completeness of the specification and consistency between artifacts.

Some of the design artifacts can be represented in UML.

One of the answers is:

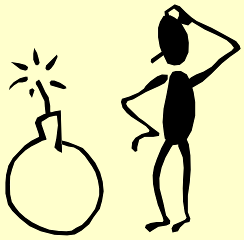
Design artifacts can be structured in various views and levels of granularity.

Here is a Little Test For You: How Do You Specify System Behavior?

a) Create scenarios

b) Create sequence diagrams

Which is correct?



To answer this question, we must realize that there is a difference between a *design artifact* and its *representation*.

The *design artifact* determines the information about the software product, and the *representation* determines how the information is presented. Some design artifacts are represented by UML diagrams, some are represented by text or by tables, and some are represented in a number of different ways.

A useful specification of a software system is based on precisely defined design artifacts, rather than on diagrams.

Scenario[1)] describes the system behavior. The scenario can be represented in a number of ways, for example, by a sequence diagram, a collaboration diagram, a statechart, an activity diagram, state transition table or by plain text.

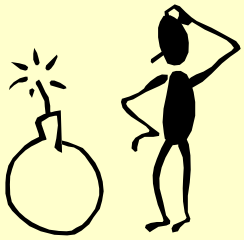
1) Later in the text the term *scenario* is substituted by more precise terms of *interaction model* and *lifecycle*.

NAVISION software

Here is a Little Test For You: How Do You Specify System Behavior?

a) Create scenarios
b) Create sequence diagrams

Which is correct?



To answer this question, we must realize that there is a difference between an *information* about the software system and its *representation*.

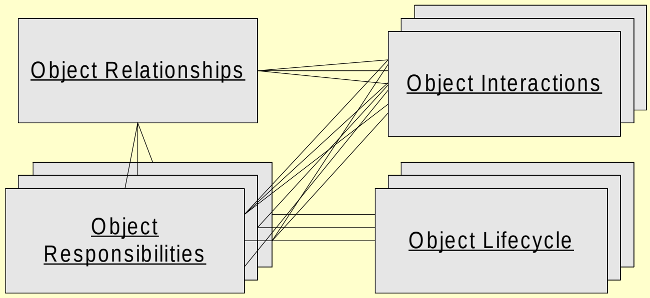
The *design artifact* determines the information about the software product, and the *representation* determines how the information is presented. Some design artifacts are represented by UML diagrams, some are represented by text or by tables, and some are represented in a number of different ways.

A useful specification of a software system is based on precisely defined design artifacts, rather than on diagrams.

Scenario[1)] describes the system behavior. The scenario can be represented in a number of ways, for example, by a sequence diagram, a collaboration diagram, a statechart, an activity diagram, state transition table or by plain text.

1) Later in the text the term *scenario* is substituted by more precise terms of *interaction model* and *lifecycle*.

Example: Artifacts Specifying a Subsystem



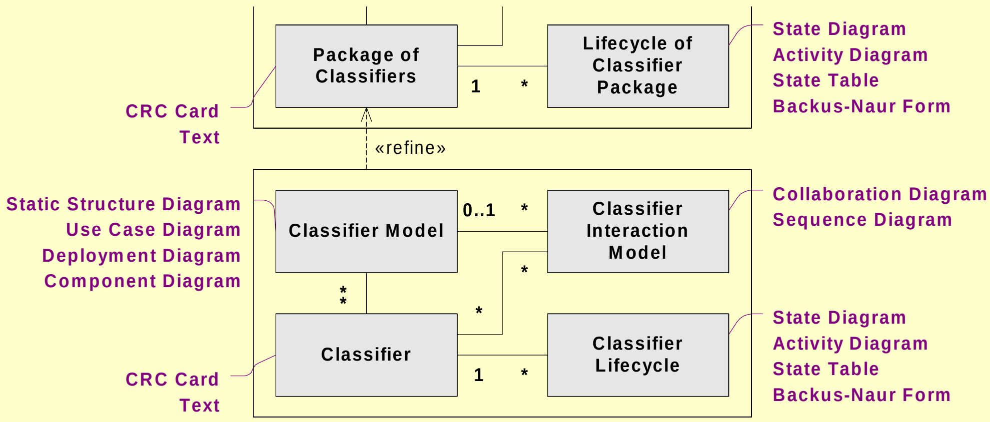
A subsystem (an instantiable package of objects) can be described by static relationships between objects, dynamic interactions between objects, object responsibilities and object lifecycles. Each of these artifacts can be represented by UML diagram or by text.

The same structure can also be used to specify packages of other UML classifiers.

UML classifiers are

- classes and objects (with various stereotypes)
- subsystems
- components
- interfaces
- nodes
- use cases

THE PATTERN OF DESIGN ARTIFACTS

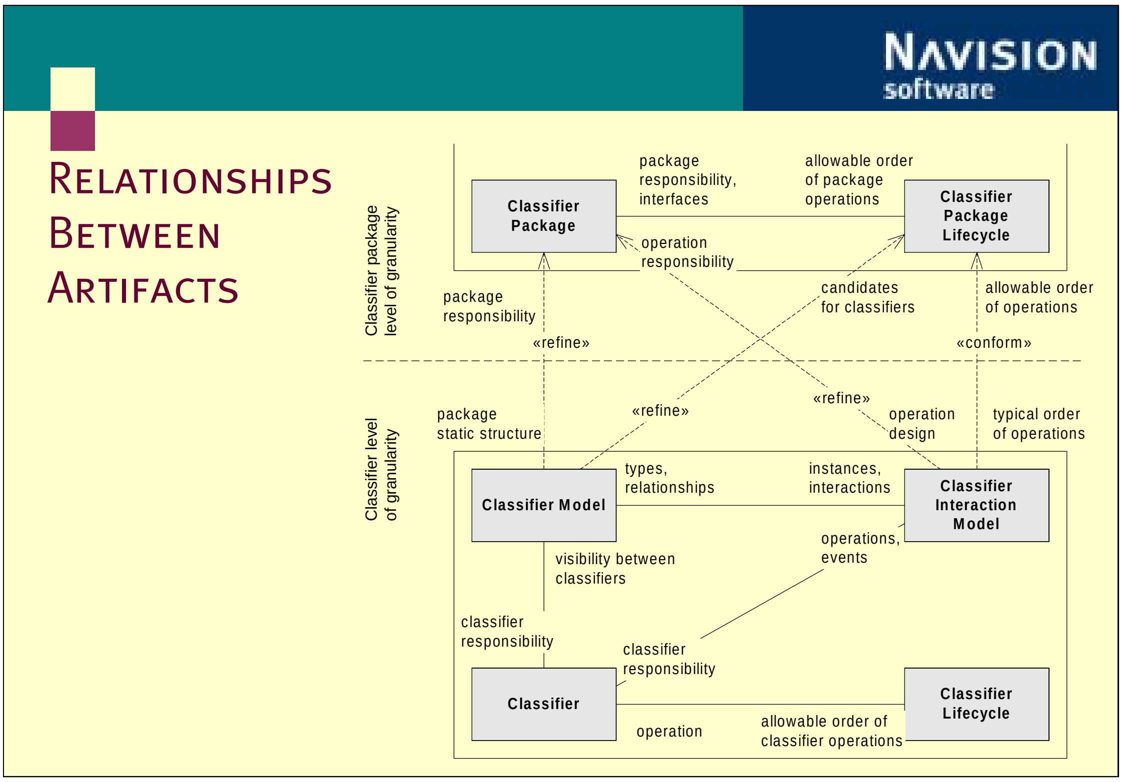


The *classifier model* specifies static relationships between classifiers. The classifier model can be represented by a set of static structure diagrams (if classifiers are subsystems, classes or interfaces), a set of use case diagrams (if classifiers are use cases and actors), a set of deployment diagrams (if classifiers are nodes) and a set of component diagrams in their type form (if classifiers are components).

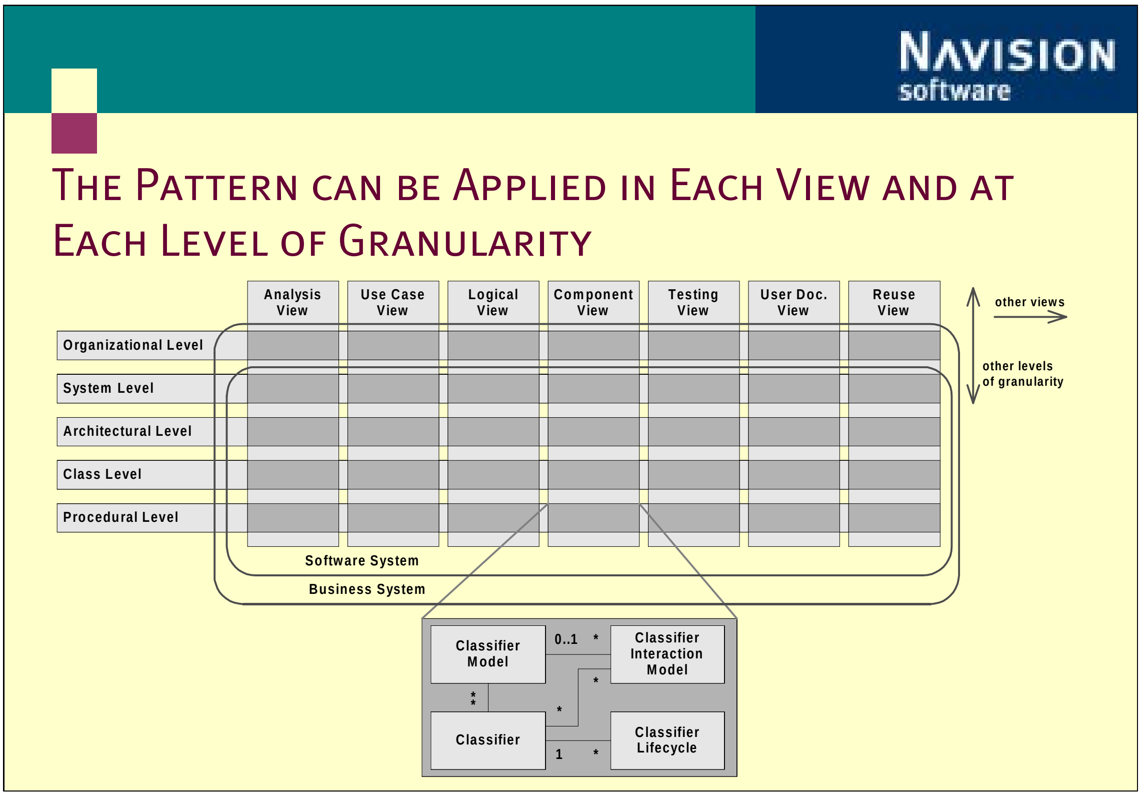
The *classifier interaction model* specifies interactions between classifiers. The classifier interaction model can be represented by sequence diagrams or collaboration diagrams.

The artifact 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 lifecycle* specifies classifier state machine and dynamic properties of classifier interfaces (for example, the allowable order operations and events).



The notation in this Figure is modified UML. For better clarity, dependencies were adorned by role ends. In the UML metamodel version 1.1, dependencies do not have role ends. However, the role ends can be specified as tags of dependencies. Please see my web page for details.



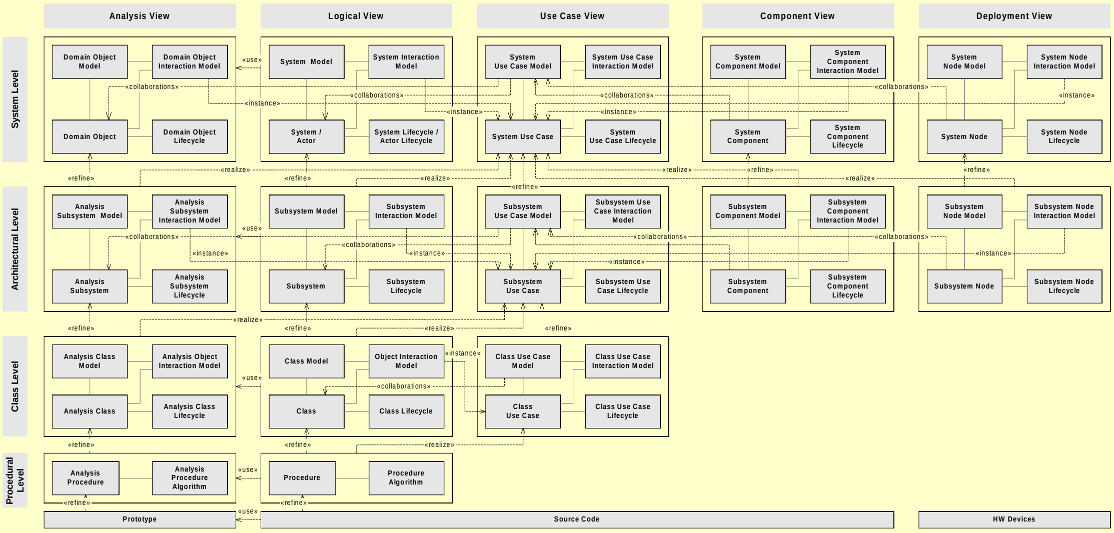
At each level of granularity and in each view, a software product can be described by classifier relationships, interactions, responsibilities and lifecycles.

This is the key point of this presentation.

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 *use case view* identifies collaborations of the system, subsystems, classes, components and nodes with actors. The *logical view* describes the logical structure of the product in terms of subsystems and classes, their responsibilities, relationships and interactions. 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 *testing view* specifies the design of tests. The *user documentation view* specifies the design of Online Help and user documentation. The *reuse view* specifies reusable elements at all levels of granularity.

The *system level* of granularity describes the context of the system, the responsibility of the system being designed and the other systems that collaborate with it. 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.

Application of the Pattern



Please see the text on the previous and the next page.

Figure on the left shows the pattern applied at four levels of granularity. The only exception in the symmetrical structure is the procedural level, which does not contain the *procedure model* (relationships between procedures) and 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 *logical view* describes the logical structure of the product. The *system, subsystem* and *class models* specify static relationships between systems, subsystems and classes. The *system, subsystem and object interaction models* describe interactions between systems, subsystems, and objects. The artifacts *system, subsystem* and *class* specify responsibilities, roles and static properties of system, subsystem and class interfaces (for example, operations and events with preconditions and postconditions). The *system, subsystem and class lifecycles* specify behavior and dynamic properties of the interfaces, for example, the allowable order of operations and events.

The *use case view* identifies collaborations of the system, subsystems and classes. The *use case model* describes static relationships between use cases and actors. The *use case interaction model* specifies typical sequences of use case instances. The *use case* specifies static properties of the collaboration, for example, the goal and pre- and postconditions. The *use case lifecycle* specifies dynamic properties of the collaboration, which is the system, subsystem and class behavior within the scope of the collaboration. Instances of use cases are specified in the *system, subsystem and class interaction models*, please see the dependency «instance». Realizations of use cases are design artifacts at lower level of granularity, please see the dependency «realize».

The *analysis view* describes design suggestions in terms of analysis objects, their responsibilities, relationships and interactions at various levels of granularity. 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.

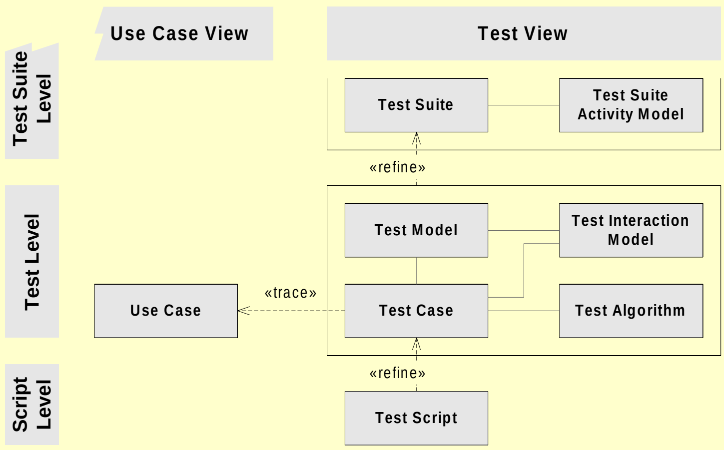
The *deployment view* describes the physical structure of the system in terms of hardware devices, their responsibilities, relationships and interactions. The *node model* specifies static relationships between the nodes and node instances, for example, hardware connections. The *node interaction model* describes interactions between node instances. The *node* specifies node responsibilities, roles and static properties of nodes. The *node lifecycle* specifies states and state transitions of the node.

The *node interaction model* represents interactions between node instances, without it being necessary to specify actual objects that send or receive messages.

The *node lifecycle* represents node states without it being necessary to specify how they are implemented.

The *component view* describes the implementation structure of the product in terms of software modules, their responsibilities, relationships and interactions.

The Pattern Applied to the Test View

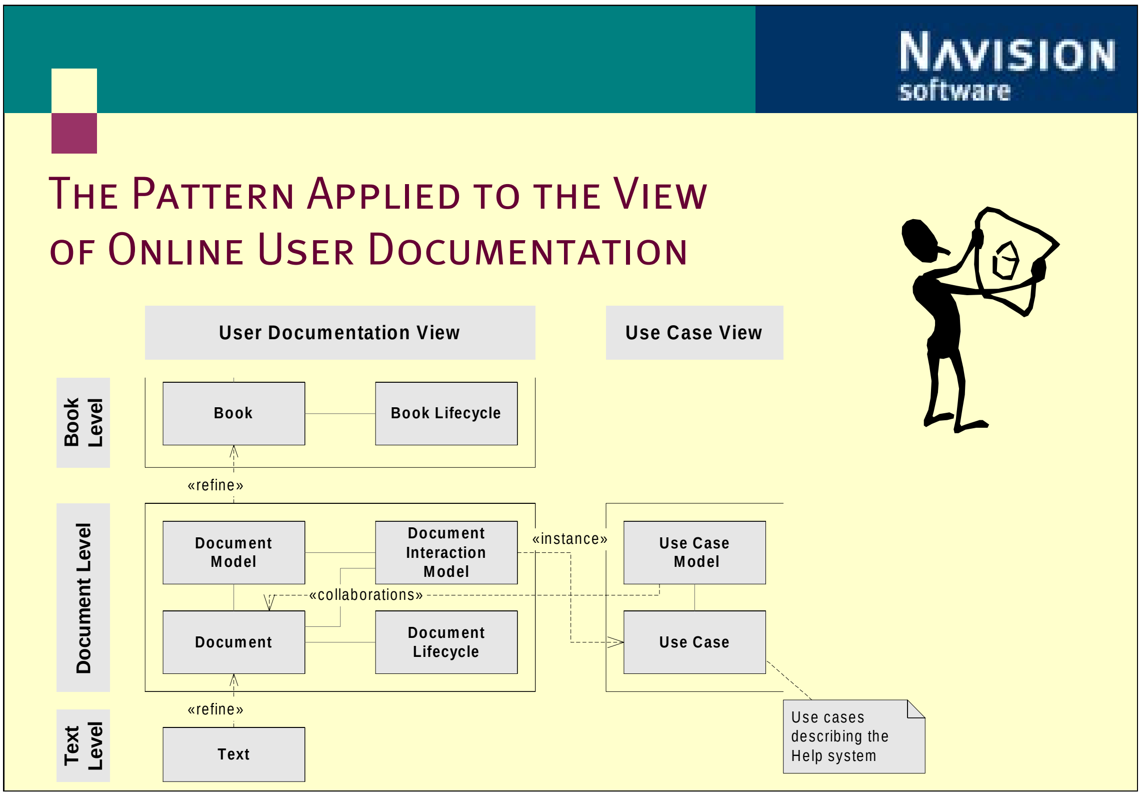


The *test view* describes the design of the software tests at different levels of granularity.

Design artifacts 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 lifecycle describing the test algorithm).

Test artifacts can be described at various levels such as the *test suite level*, the *test level* and the *test script* level. Design 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). The test specification can be extended to other levels of granularity.

The dependency with the stereotype «trace» indicates that test cases can be based on use cases.

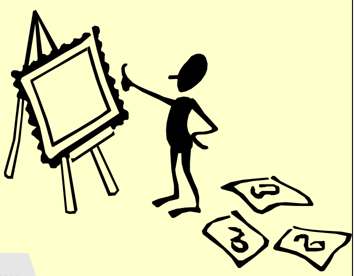


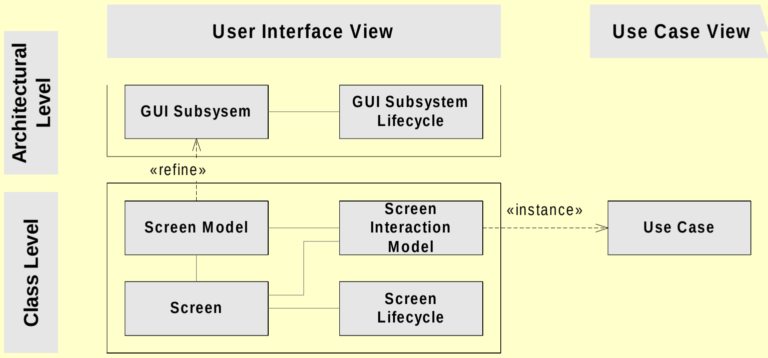
Documents (pages in online help or Internet pages) are shown as stereotyped components in UML. Design artifacts for designing user documentation are the *document model* (static relationships between documents), the *document interaction model* (typical scenarios that arise in searching for particular information), the *document* (short descriptions of their purpose and contents) and the *document lifecycle* (if the document has behavior). The *use case model* specify typical cases how to use the online help or user documentation and *use cases* are generalized searching scenarios.

Design artifacts for user documentation can also be described at various levels: the book level, the document level and the text level.

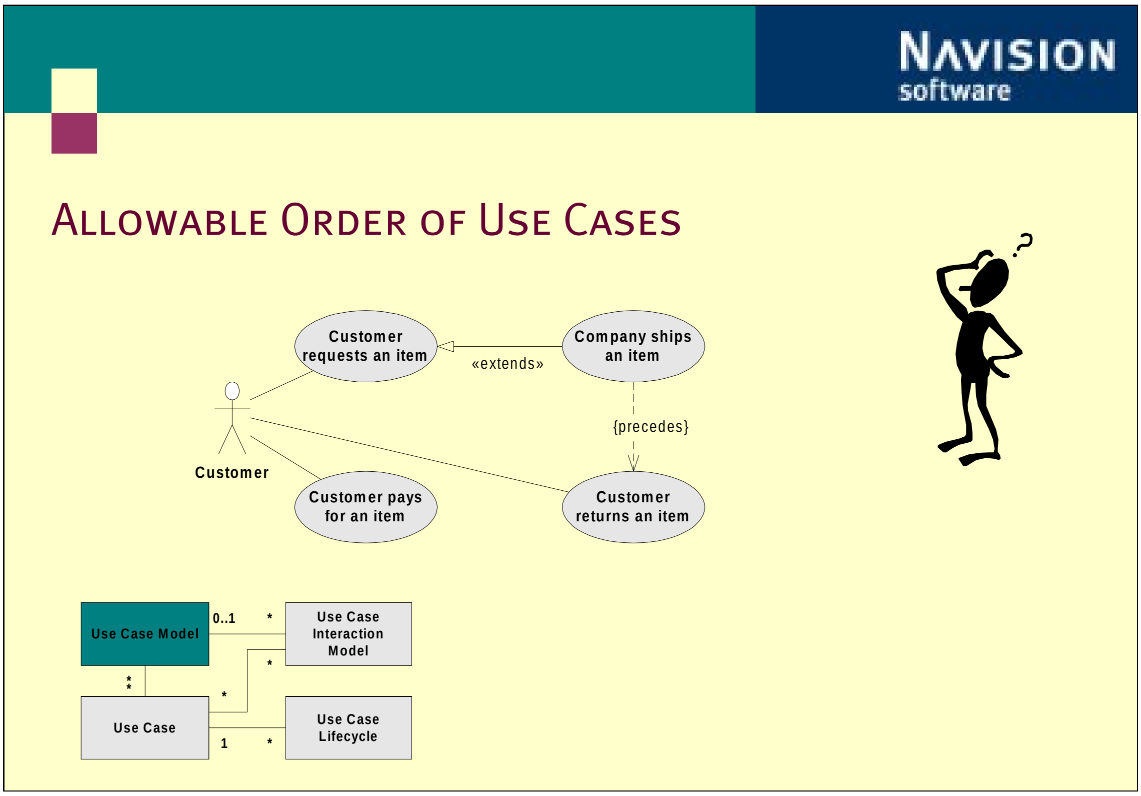
.

The Pattern Applied to the User Interface View





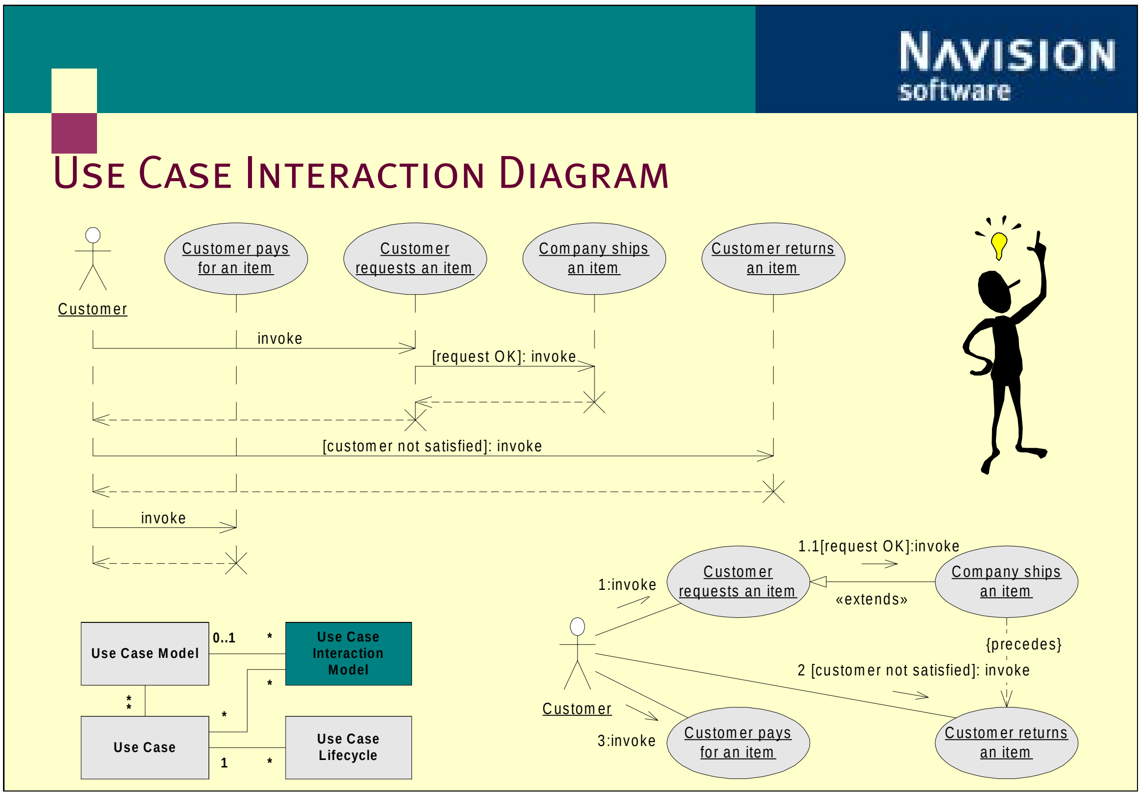
Screens (windows) can be shown as stereotyped classes in UML. Design artifacts for designing user interface are the *screen model* (static relationships between screens), the *screen interaction model* (typical sequences of activation of screens), the *screens* (their responsibilities with sample drawings, for example), and the *screen lifecycle* (if the screen has behavior). The dependency with the stereotype «instance» indicates that screen interactions are instances of use cases. The artifact *GUI subsystem* specifies responsibilities of GUI subsystems in cases in which the system interacts with groups of actors whose user interface requirements are so distinct that different user interface subsystems are required.



UML use case diagram cannot easily express that a customer first requests an item, then a company ships an item, and then the customer pays for an item.

One of the solutions is to use constraints {precedes}, or dependencies «precedes» between use cases. Similar relationships exist in OML[1)] (OPEN modeling language). However, the diagram with constraints or dependencies is still a static structure diagram, not a dynamic scenario.

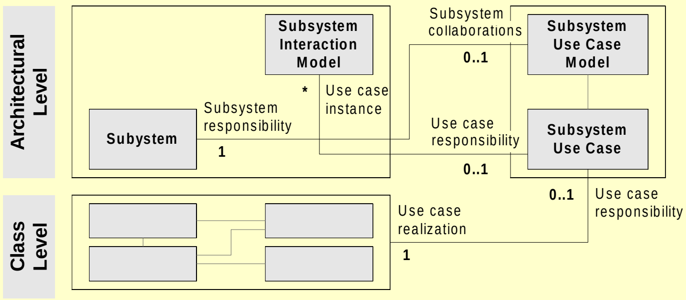
1) Please see http://www.csse.swin.edu.au/cotar/OPEN/OPEN.html

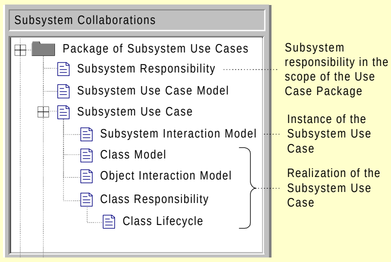


Use case interaction diagrams are sequence and collaboration diagrams in which classifier roles are use case roles. The use case interaction diagram conforms to the UML metamodel, but it is not explicitly mentioned in the UML Notation Guide

Use case interaction diagram is the only UML diagram that can represent scenarios consisting of use case instances. The messages *invoke* represent constructors of the use cases and they map to the signals from the actors to the use cases. They can also be named according to the first operation in each use case, such as *invoke request*, *invoke shipment* and *invoke payment*. Except for these messages, the use case interaction diagram can show other messages exchanged between the actor and the system and describe complete use case conversations (The figure above shows only constructors of use cases).

ARTIFACTS CAN BE STRUCTURED ACCORDING TO COLLABORATIONS





Structuring design artifacts according to collaborations (their relationships to a use case) is useful for understanding the system functionality in a particular context.

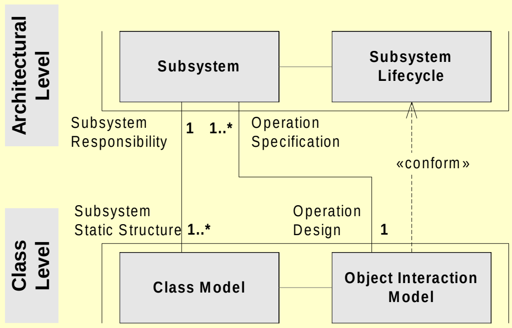
UML 1.1 does not have any specific symbol for *collaboration*. Therefore, on this poster I am showing collaborations as use cases, because semantics of use cases and collaborations are very similar.

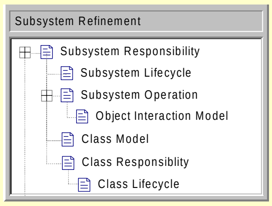
Relationships used in this structure are shown as dependencies with the stereotypes «instance», «realize» and «collaborations». These dependencies are refined to associations, because associations are more descriptive than dependencies. Unlike dependencies in UML, associations can be adorned with role ends.

Figure shows an example, in which the *subsystem use case model* can contain several packages of use cases. Each one of these packages is linked to the *subsystem*, which specifies the system responsibility in the scope of this use case package. The responsibility of each use case in the package is specified in the artifact *subsystem use case*. Instances of these use cases are shown in the *subsystem interaction model*, and their realizations are specified as a cluster of four design artifacts at the class level in the logical, implementation and deployment views.

The associations between artifacts are on the left, an example of the repository structure is on the right.

Artifacts Can be Structured According to Refinement between Levels of Granularity





Structuring design artifacts according to refinements is useful for understanding the overall structure and functionality of the system, component or class.

These relationships are shown as dependencies with a stereotype «refine». These dependencies are refined to associations between artifacts.

Figure shows an example in which the artifact *subsystem* specifies subsystem responsibilities and subsystem interfaces. The *class model* specifies the static structure of the subsystem, and the *object interaction model* specifies the design of each operation in the subsystem interface in terms of object interactions. The dependency «conform» indicates that the operation design has to match the dynamic properties of the subsystem interface specified in the subsystem lifecycle.

The associations between artifacts are on the left. An example of the repository structure is on the right.

Navision software

Is This a Heavyweight or a Lightweight Structure?

The minimal set of artifacts must include

- System responsibility
- Code

The pragmatic set of artifacts must include

- System responsibility
- Object interaction model
- Code

The typical set of artifacts must include

- . . . (See the pages with the Fusion, Objectory and Shlaer-Mellor methods.)

The pattern enables you to customize the size of the specification, so it matches the size of the problem and stays consistent.

The pattern does not force you to create design artifacts you do not need. It only gives you an overview over the completeness of the specification.

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.

NAVISION
software

WHAT ABOUT THE DEVELOPMENT PROCESS?

It should be clear that this poster specifies the static structure of design artifacts.

Various development processes specify interactions between design artifacts and the order in which they are created and completed.

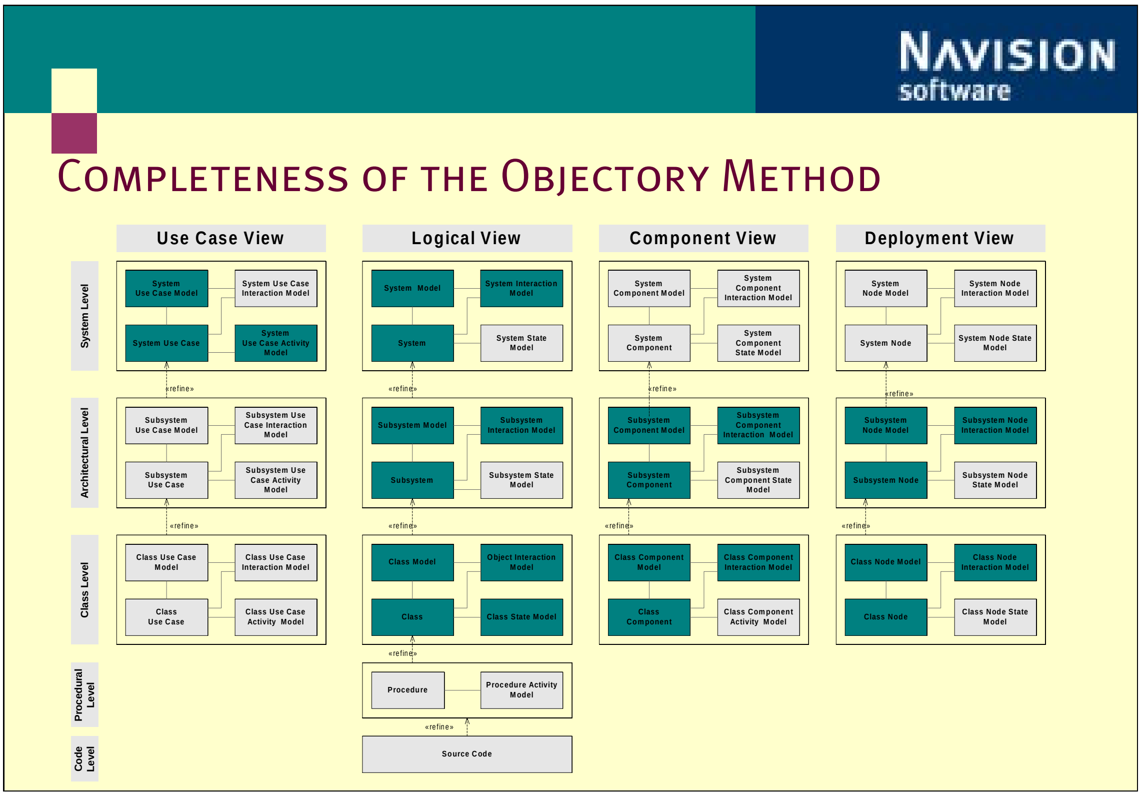
Use Case 1: Waterfall Process (top-down)
Use Case 2: Prototyping (bottom-up)
Use Case 3: Iterative and Incremental Process
Use Case 4: . . .(depends on your concrete problem)

To specify a software development process, it is useful to consider design artifacts as objects and evolution as collaborations between objects.

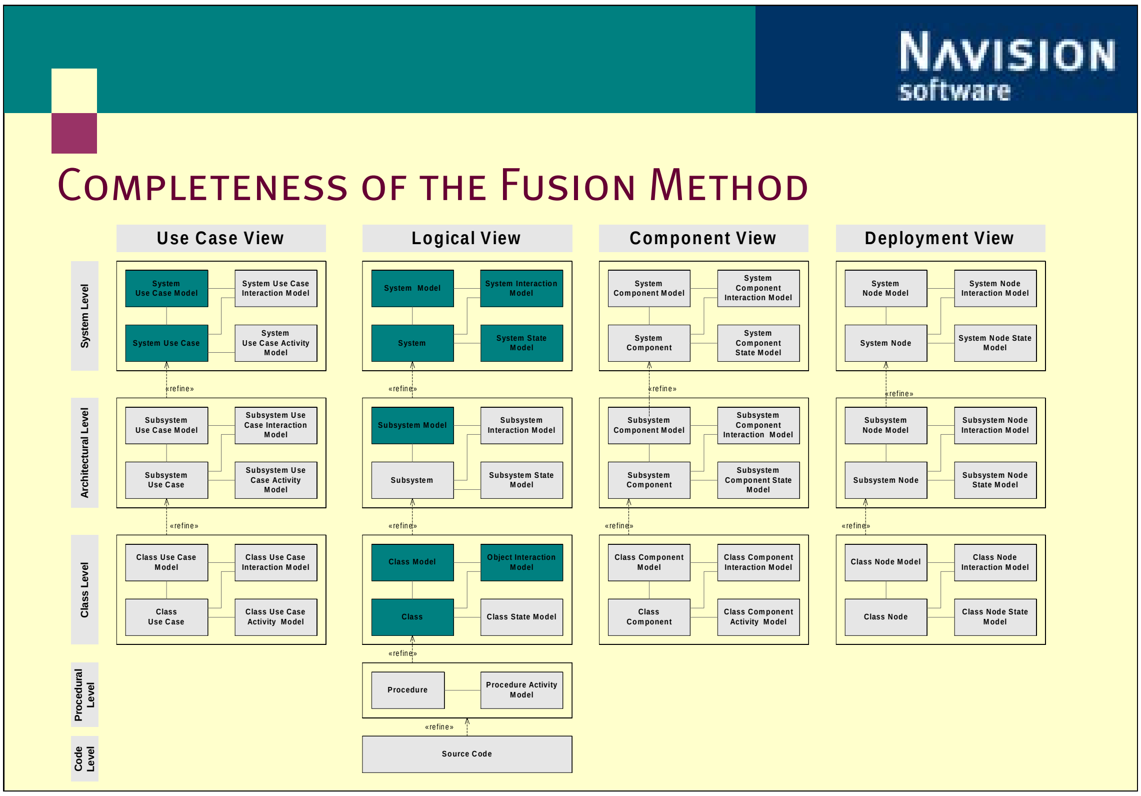
Design artifacts have *attributes* such as name, representation, owner, project and increment identification, in addition to attributes, such as who created and modified the artifact instance and when.

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 the complexity of a development process in a better way than a description based on workflow. It is also easier to customize it during the process.

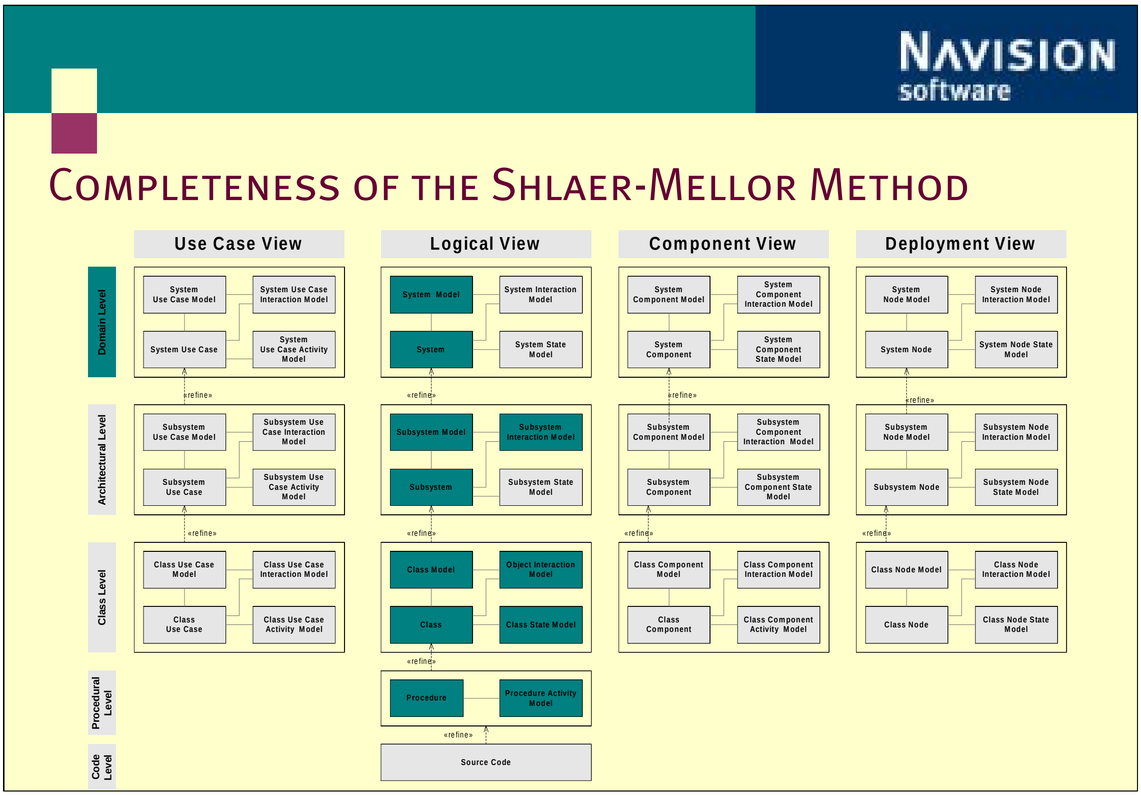
I have discussed my experience with such a process definition at OOPSLA’97. The paper is available on Navision web page.



The artifacts of the Objectory method are structured in the use case, logical, deployment, implementation and process views, and at the 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 lifecycles with the exception of the use case activity model and the class state model. The design artifacts are structured according to their relationships to use cases (in other words, according to their collaborations with external actors).



The Fusion method is a method with a succinct and consistent system of artifacts that is orthogonal, which means that one fact about the product is stated only in one place.
The Fusion method focuses on artifacts in the logical view at the 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 lifecycle (*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 lifecycles except of the system lifecycle.
Design artifacts are structured according to the refinement between levels of granularity.
The new Fusion Engineering process (also known as Team Fusion) also produces use cases and a use case model.



The Shlaer-Mellor method has one of the best systems of design artifacts. 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 (SM) is focused on the logical view, and therefore the method does not produce any artifacts in the use case, component or implementation views.
The Shlaer-Mellor method does not produce any artifacts at the system level. The method recognizes an extra domain level 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 lifecycle (*state transition diagram* and *class structure chart* in SM). At the procedural level, Shlaer-Mellor produces the procedure (*action specification* in SM) and the procedure algorithm (*action data flow diagram* in SM).

Can You Help Me?

In spite of the fact that this pattern can solve some problems that arise when using UML, there are several interesting unresolved issues.

1. The term *design artifact*
2. The term *model*
3. The *orthogonal system* of design artifacts

1. The term *design artifact* is perhaps not the best one to use. However, the terms *deliverable, model* or *process product* have drawbacks as well. I want to stress that a design artifact is a piece of design information about the *product*. Design information does not include, for example, a consistency check, a process phase, a coding activity, or meeting minutes. However, these terms are sometimes called *artifacts*. Please advise what can be done about this apparent ambiguity?

2. I use the term *model* as a synonym for *static structure*. For example, the artifact called *class model* specifies static relationships between classes. This definition conforms to the Objectory method.

However, it might be more convenient to use the word *model* to mean the *cluster of design artifacts*. For example, the *class model* of the subsystem would mean the class relationships, interactions, responsibilities and lifecycles.

If we choose the latter definition of the term model, then what do we call the design artifact specifying static relationships between classes? Suggestions?

3. 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. Both statecharts and interaction diagrams specify messages between objects. Can we find a rule for choosing an orthogonal set of UML diagrams? What is the minimal complete set of design artifacts?