NAVISION software

STRUCTURING DESIGN DELIVERABLES WITH UML

«UML»'98, Mulhouse, France, June 3-4, 1998

Pavel Hruby
Navision Software a/s
ph@navision.com

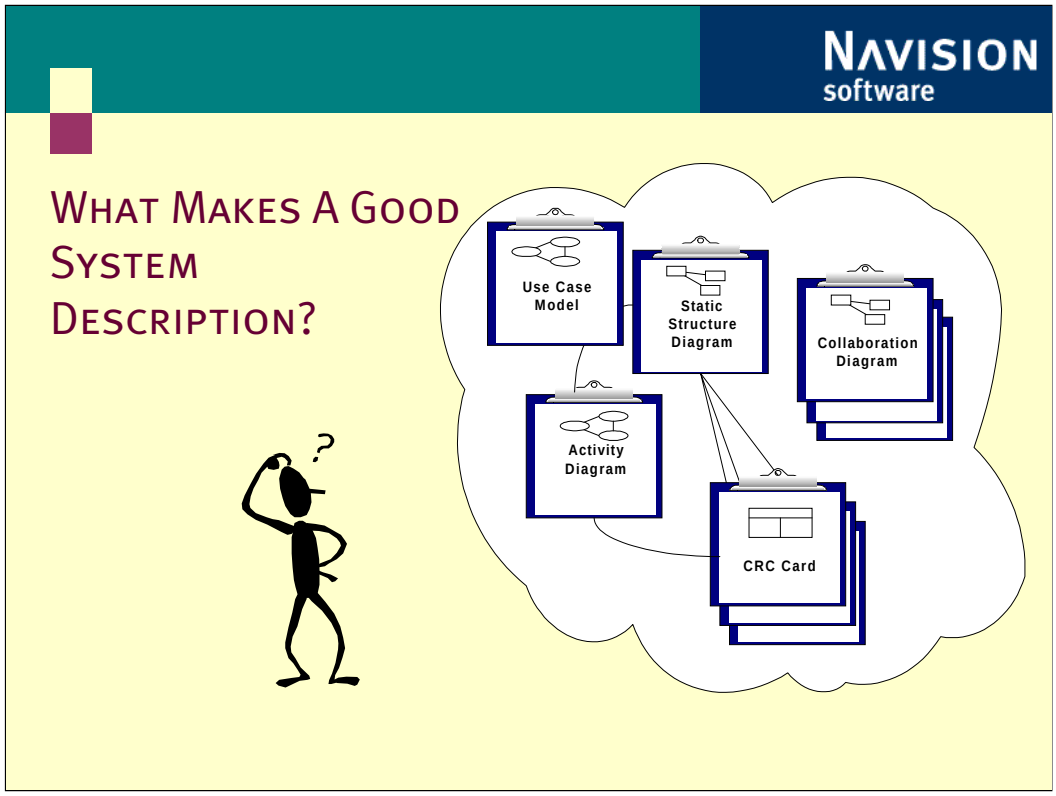
NAVISION
software

NAVISION SOFTWARE

Strategic provider of efficient accounting and business management solutions.

Address: Frydenlunds Allé 6
2950 Vedbæk
Denmark

http://www.navision.com



UML defines a standard notation for object-oriented 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 software models or what the relationships between various models are.

NAVISION software

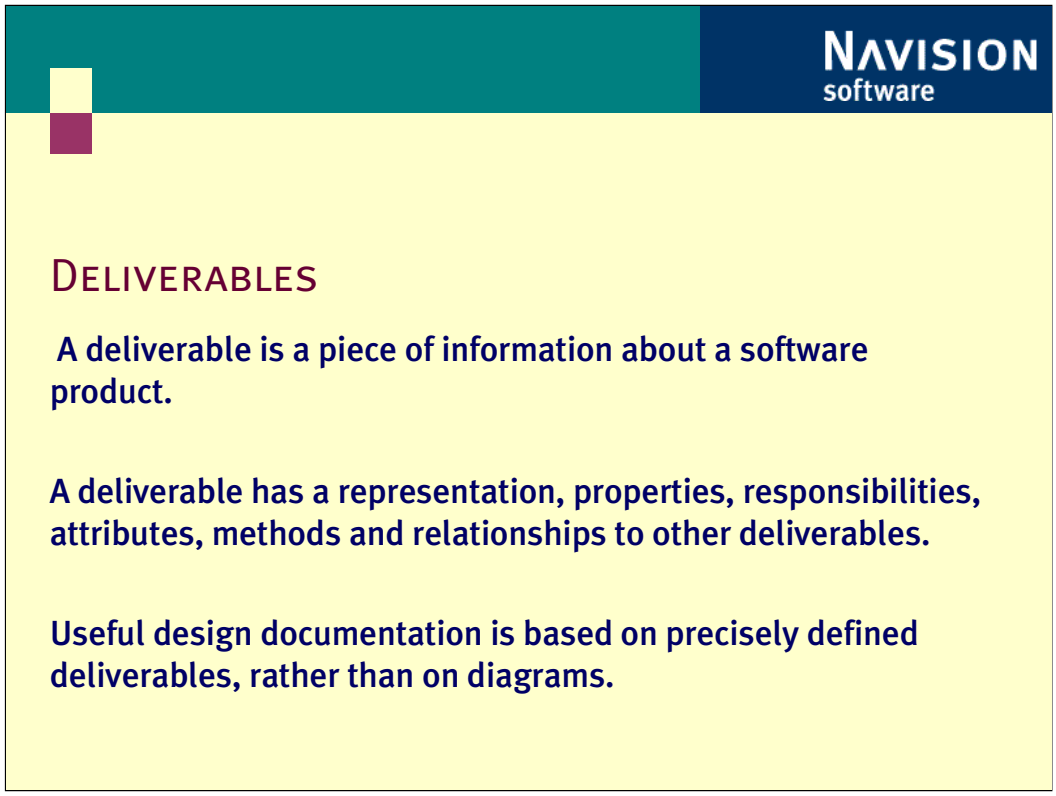
EXAMPLE:
HOW TO SPECIFY SYSTEM BEHAVIOR

a) Create scenarios

b) Create sequence diagrams

What is correct?

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.



The word *deliverable* is perhaps not the best one to use.

However, the terms *model*, or *artifact* have drawbacks as well.

If you have other suggestions about how to refer to “a unit of information about software system,” please send me an e-mail at ph@navision.com

NAVISION software

Diagrams are Representations of Deliverables

Interaction model is represented by

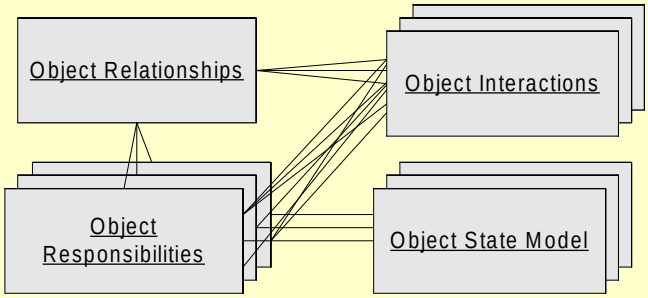
- Sequence Diagram
- Collaboration Diagram

State model is represented by

- Statechart Diagram
- Activity Diagram
- State Transition Table

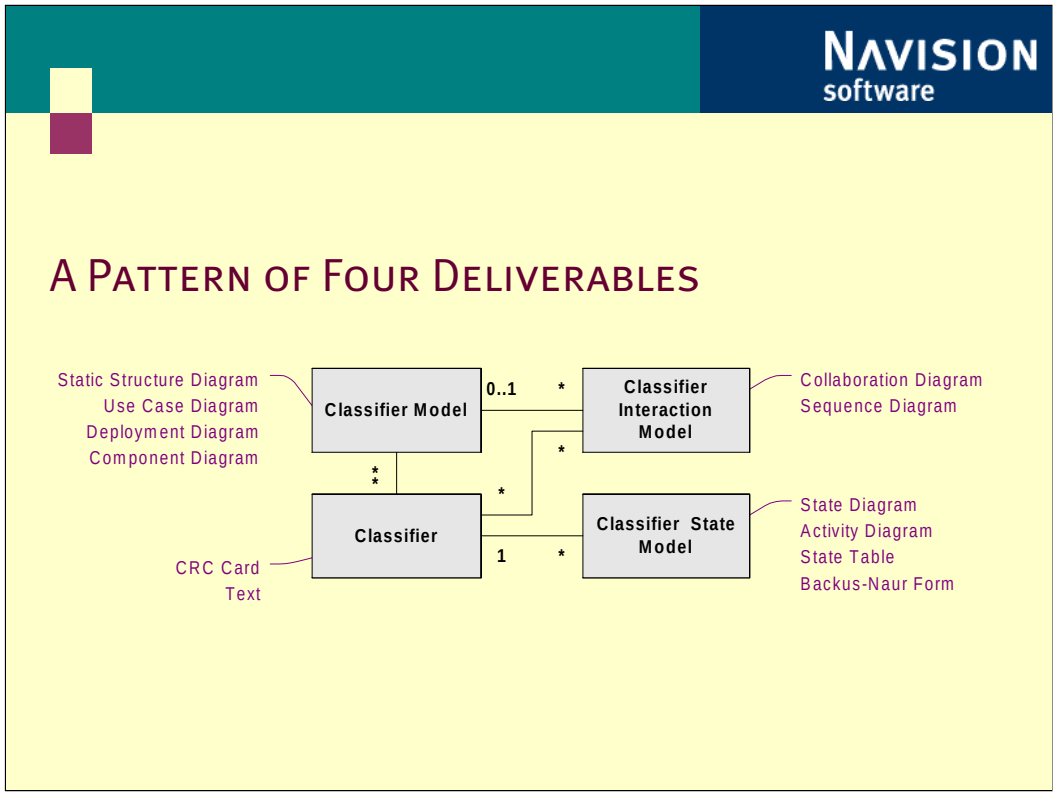
NAVISION software

EXAMPLE: LOGICAL DESIGN OF A SUBSYSTEM



A deliverable specifying object relationships is linked to several deliverables specifying object interactions. All of these deliverables are linked to deliverables specifying object responsibilities. A deliverable specifying object responsibilities is linked to a deliverable specifying object state model.

The same structure can also be used with other UML classifiers, such as classes, subsystems, components, interfaces, nodes and even with use cases.



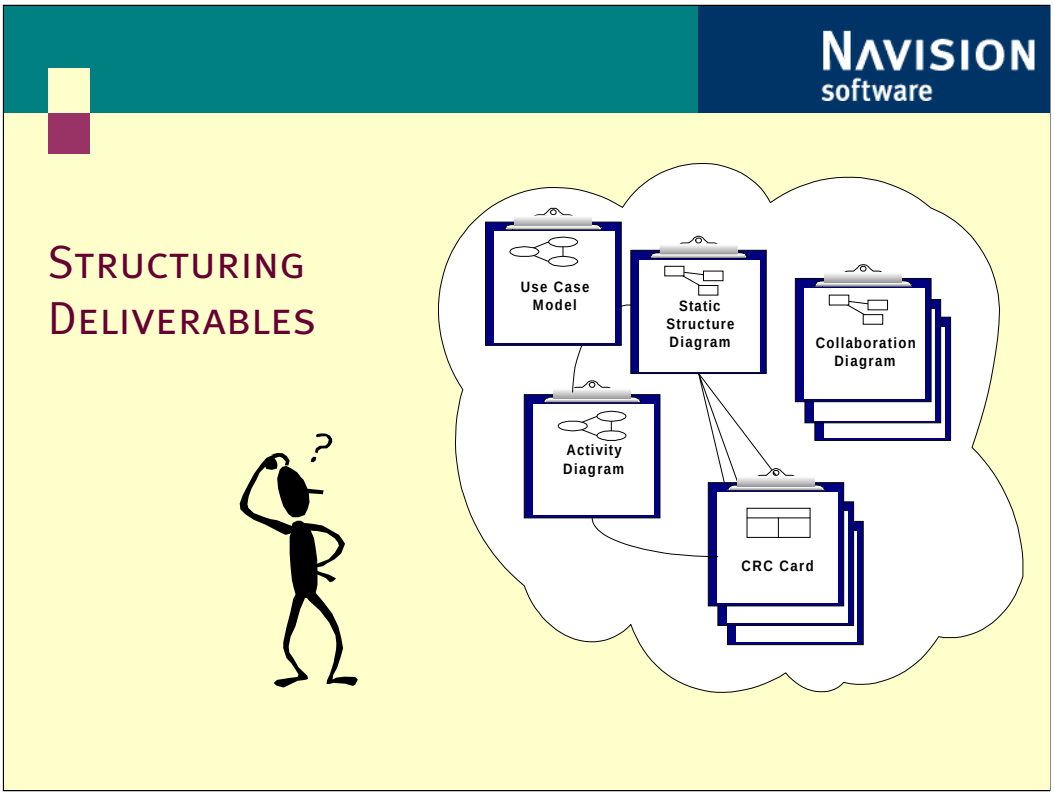
The *classifier model* specifies static relationships between classifiers. The classifier model can be represented by a set of static structure diagrams (if classifiers are subsystems, classes or interfaces), a set of use case diagrams (if classifiers are use cases and actors), a set of deployment diagrams (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 interaction diagrams: sequence diagrams or collaboration diagrams.

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).

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*.

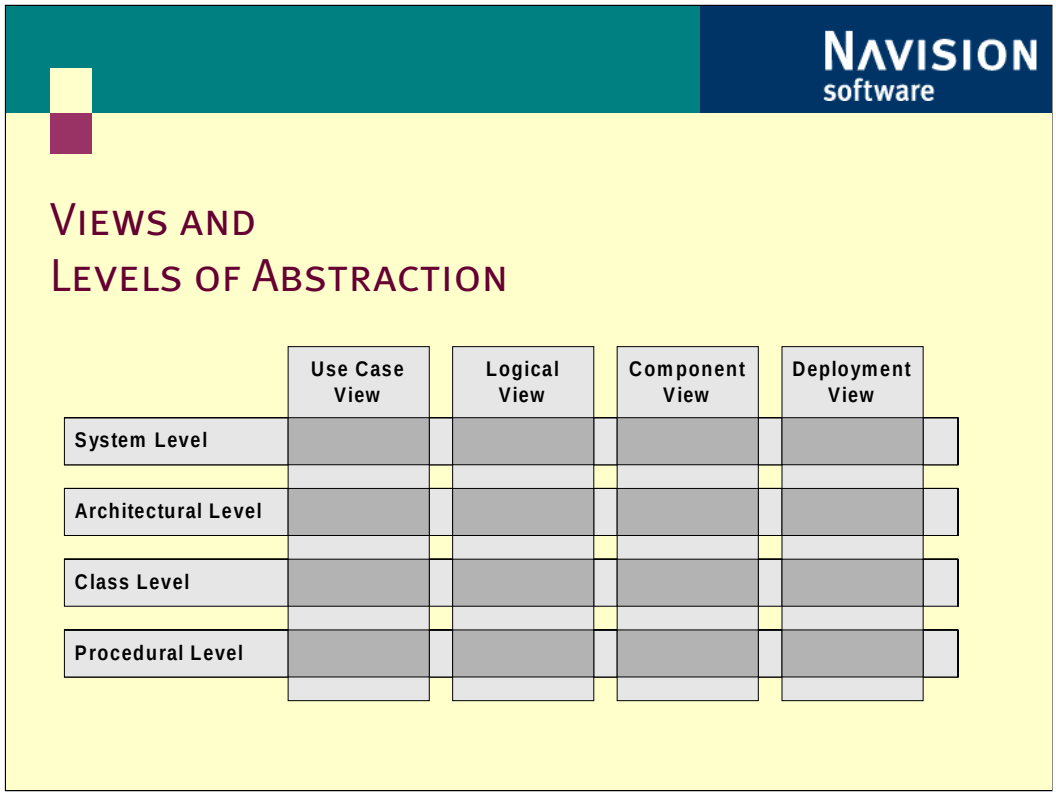


Some of the deliverables can be represented in UML.

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.

One of the answers:

Deliverables can be structured in various views and levels of abstraction.



UML is intended preferably to be used in in use case, logical, component and deployment views. However, the software product can be described in other views, such as the test view, the database design view, the user interface view and the user documentation view.

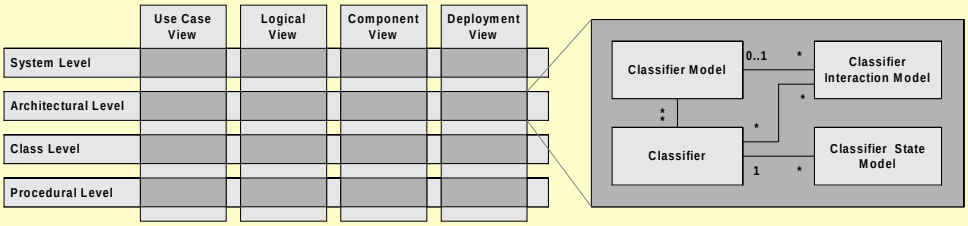
Typically, the software product is described at the system, architectural, class and procedural levels of abstraction. The *system level* describes the context of the system. The system level specifies responsibilities of the system being designed and responsibilities of the systems that collaborate 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.

The product can also be described at other levels of abstraction, such as the tier level, the domain level, the analysis level and the business object level.

NAVISION software

APPLYING THE PATTERN

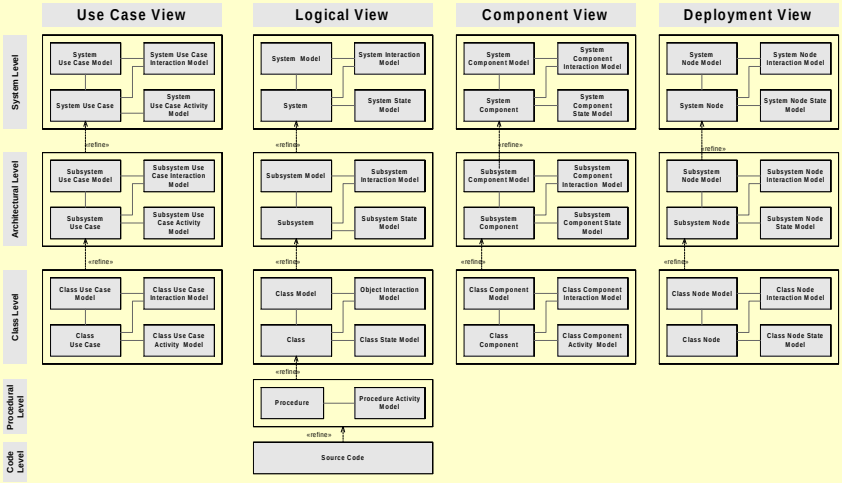
At each level of abstraction and in each view, a software product can be described by classifier relationships, interactions, responsibilities and state machines.



This is a key point of my presentation.

NAVISION software

APPLYING THE PATTERN



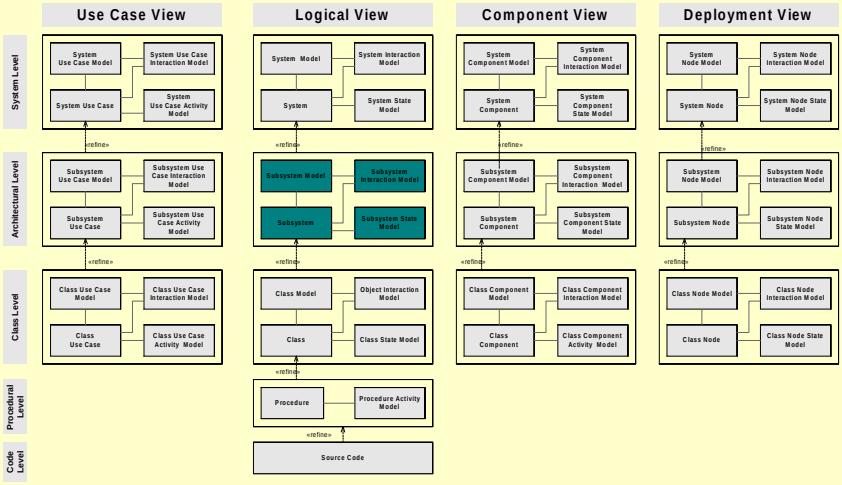
Each deliverable (gray rectangle) contains a specific piece of information about a software product.

The only exception in the symmetrical structure 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.

We have been already talking about the deliverables in the logical view and at the class level of abstraction.

NAVISION software

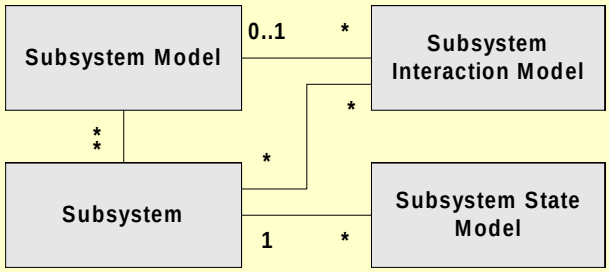
System Architecture - Logical View



The logical view and the architectural level of abstraction.

NAVISION software

SYSTEM ARCHITECTURE - LOGICAL VIEW



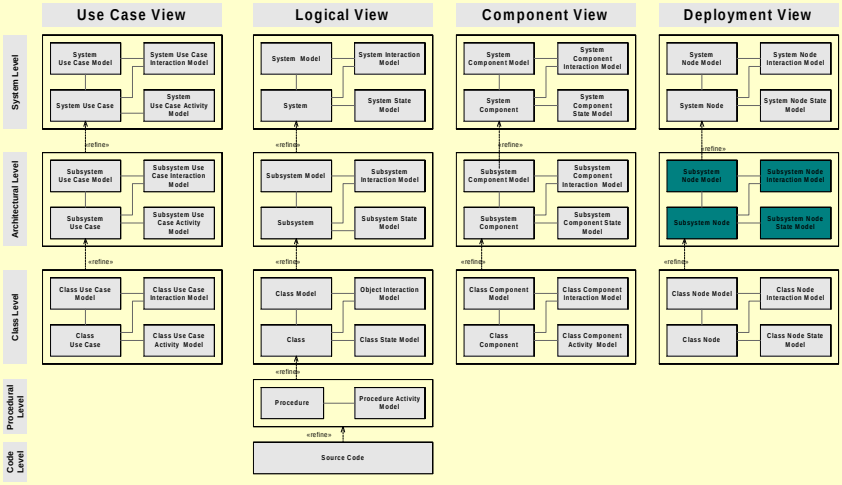
The *subsystem model* specifies static relationships between the subsystems. The *subsystem interaction model* describes interactions between subsystems. The *subsystem* specifies subsystem responsibilities, roles and static properties of subsystem interfaces (for example, a list of subsystem operations and events). The *subsystem state model* specifies behavior of the subsystem and dynamic properties of subsystem interface, for example, the allowable order of subsystem operations and events.

The subsystem interaction diagram represent interactions between subsystems, without it being necessary to specify actual objects that send or receive messages.

The *subsystem state model* specifies dynamic properties of subsystem interfaces, without it being necessary to specify in which objects they are implemented.

NAVISION software

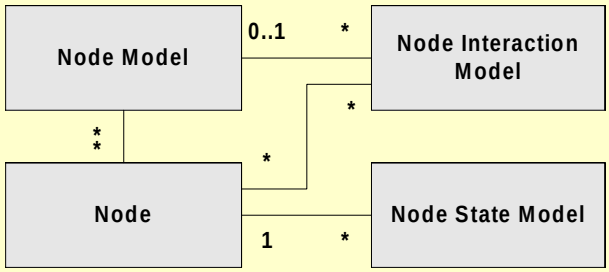
System Architecture - Deployment View



The deployment view and the architectural level of abstraction.

NAVISION software

System Architecture - Deployment View



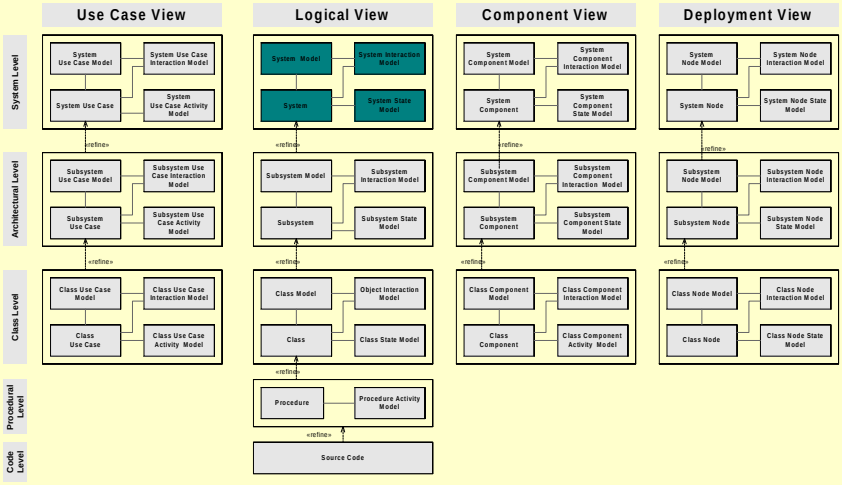
The *node model* specifies static relationships between the nodes, for example hardware connections. The *node interaction model* describes interactions between nodes. The *node* specifies node responsibilities, roles and static properties of nodes and node interfaces. The *node state model* specifies 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 state model represents node state variables without it being necessary to specify how they are implemented.

NAVISION software

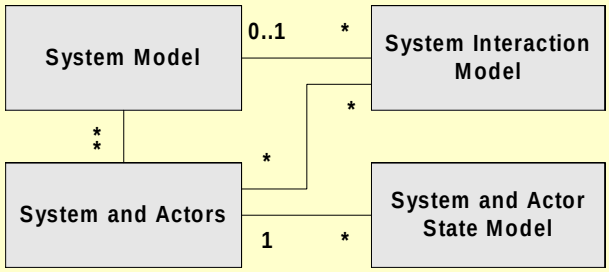
System Context - Logical View



The logical view and the system level of abstraction.

NAVISION software

System Context



The *system model* specifies static relationships between the software product and collaborating systems.

The *system interaction model* describes interactions between the software product and collaborating systems. These interactions are instances of system use cases.

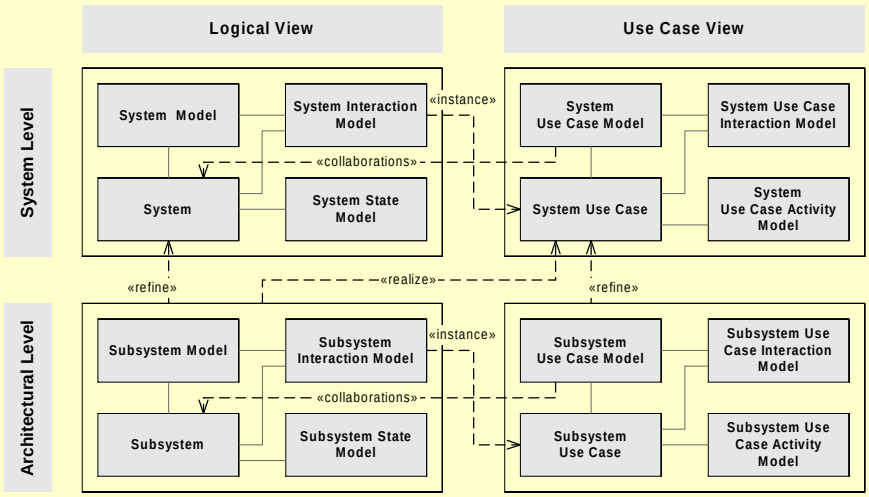
The *system* specifies system responsibilities, roles and static properties of system interfaces (for example, a list of system operations and events). Other instances of this model can represent responsibilities of external systems and actors, if they are relevant.

The *system state model* specifies behavior of the system and dynamic properties of system interface, for example, the allowable order of system operations and events.

In the Fusion method, the system state model is called *system lifecycle*.

NAVISION software

RELATIONSHIPS BETWEEN DELIVERABLES

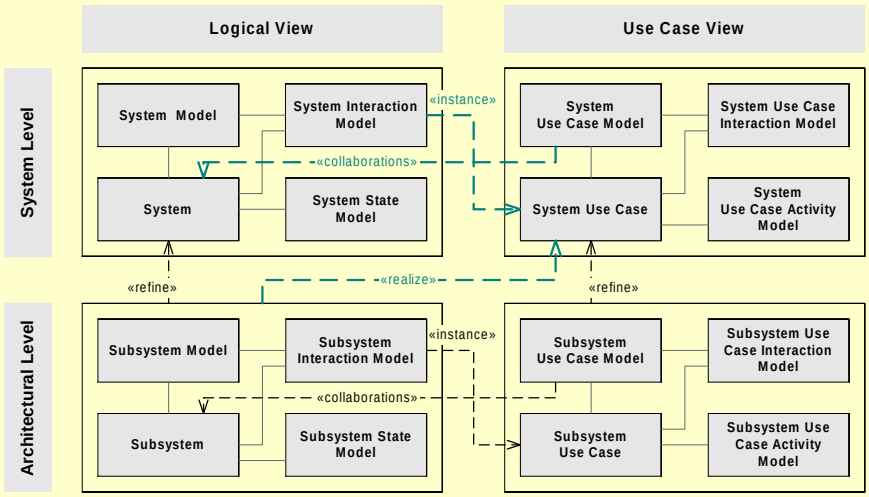


An efficient structure of design deliverables is based on relationships between them.

In well-structured design documentation, closely related information is linked together.

NAVISION software

COLLABORATIONS AT THE SYSTEM LEVEL

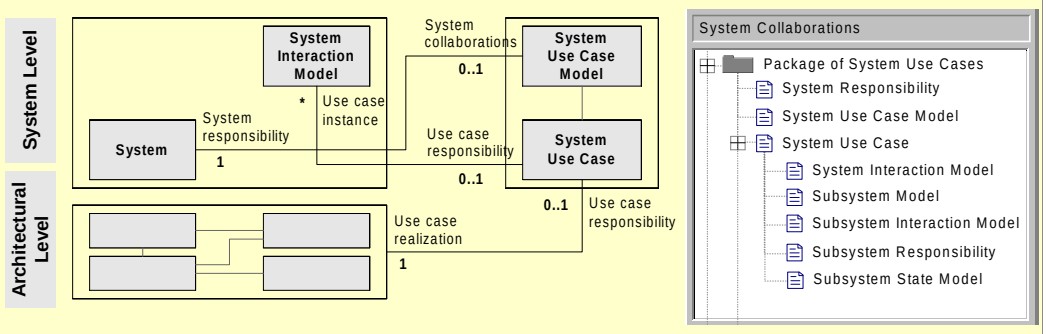


UML 1.1 does not have a symbol for collaboration. Therefore, in this presentation I assume that collaborations are specified by use cases.

In this presentation, the term *use case* in used in a wider sense than in UML. The system, subsystem, class, component and node use cases are collaborations of the system, subsystem, class, component and node with other classifiers. These other classifiers can be inside or outside the system.

NAVISION software

Collaborations at the System Level: Structuring Deliverables

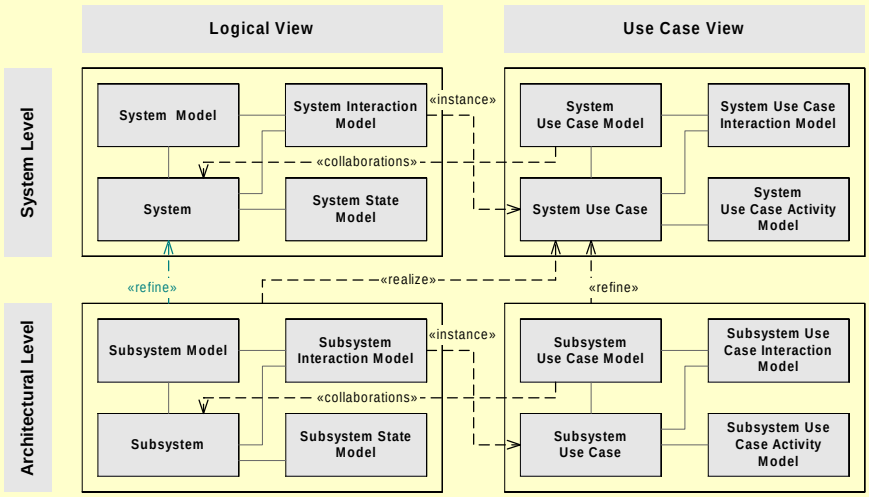


I use UML in this presentation. In UML, dependencies cannot have roles at their ends. (I hope that the next version of UML will allow dependencies with role ends!) Therefore, in this slide I replace the dependencies from the previous slide by associations.

The associations between deliverables are on the left, an example of their projection is on the right.

NAVISION software

REFINEMENT AT THE SYSTEM LEVEL

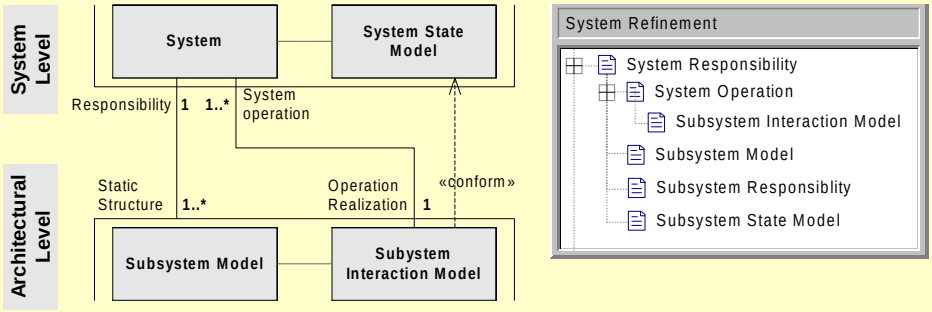


The deliverable *system* specifies responsibility of the system and static properties of the system interface (for example, a list of system operations and events).

The deliverable *system* is refined into the deliverables *subystem model*, *subystem interaction model*, *subystem state model* and *subystem*, which represent design of the system.

NAVISION software

Refinement at the System Level: Structuring Deliverables

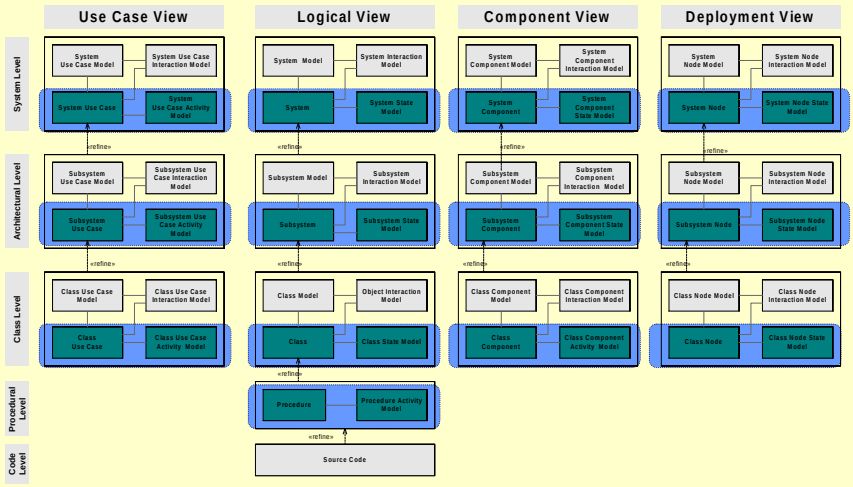


The dependency «refines» from the previous slide was “refined” into several associations.

The associations between deliverables are on the left. An example of their projection is on the right.

NAVISION software

Several Ways how to Simplify the Structure

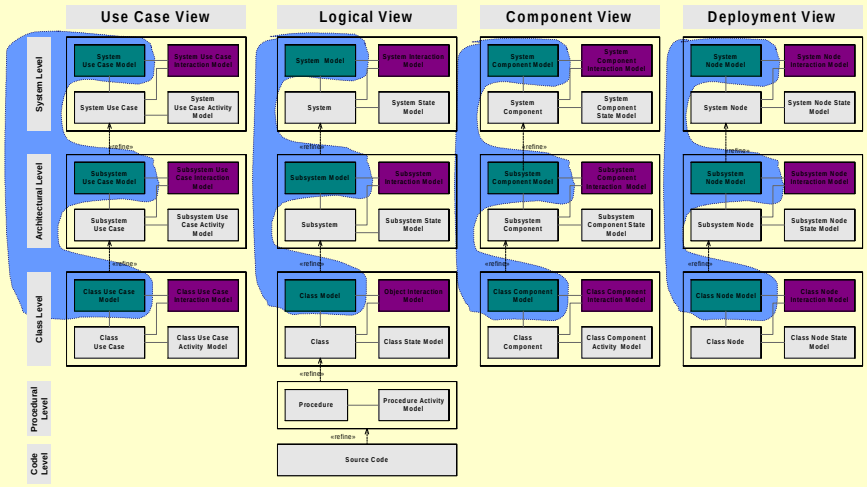


Typically, instances of deliverables are separate documents. However, there might be pragmatic reasons for creating documents containing several closely related deliverables.

For example, the deliverables *classifier* and *classifier state model* are always linked together. They can be joined into one document.

NAVISION software

Several Ways how to Simplify the Structure



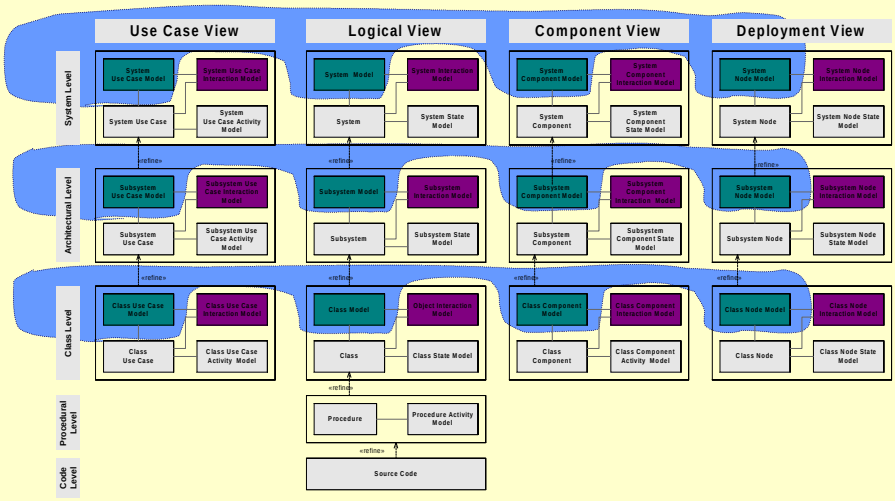
It is also possible to join system, subsystem and class use case models to one use case diagram, providing that use case levels are clearly distinguished. It is necessary to distinguish levels of use cases, because use cases at different levels are related to different deliverables.

It is also possible to create one static structure diagram containing actors, system, subsystems, classes and procedures. 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.

Interaction models (magenta color) can be joined together in the same way as the classifier models (green color).

NAVISION software

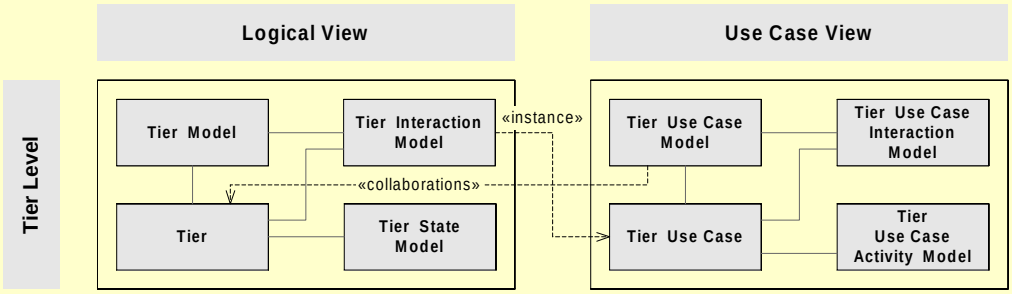
Several Ways how to Simplify the Structure



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, although the *UML Notation Guide* does not mention such a combined static structure diagram.

NAVISION software

OTHER APPLICATIONS OF THE PATTERN: LAYERED ARCHITECTURE

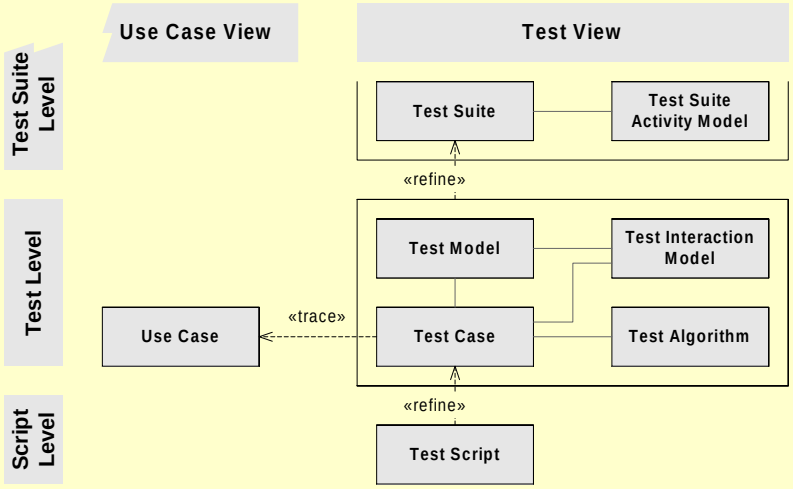


Systems with layered architecture have a *tier level* between the system level and the architectural level. The *tier* level specifies system layers, their relationships and interactions. In a layered system each layer contains subsystems and components.

The tier level contains the tier model (relationships between system layers), the tier interaction model (interactions between system layers), the tier (responsibility of the layer) and the ties state model (dynamic properties of the system layer interfaces). Models at the tier level are usually refined into models at the architectural level.

NAVISION software

Other Applications of the Pattern: Software Testing

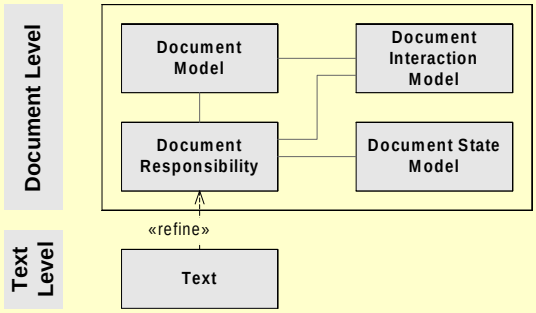


Deliverables in the test view are the test model (static relationships between tests), the test interaction model (interactions between tests), the test case (description of the test), and the test algorithm (test activity model describing the test algorithm). Test deliverables can be described at various 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).

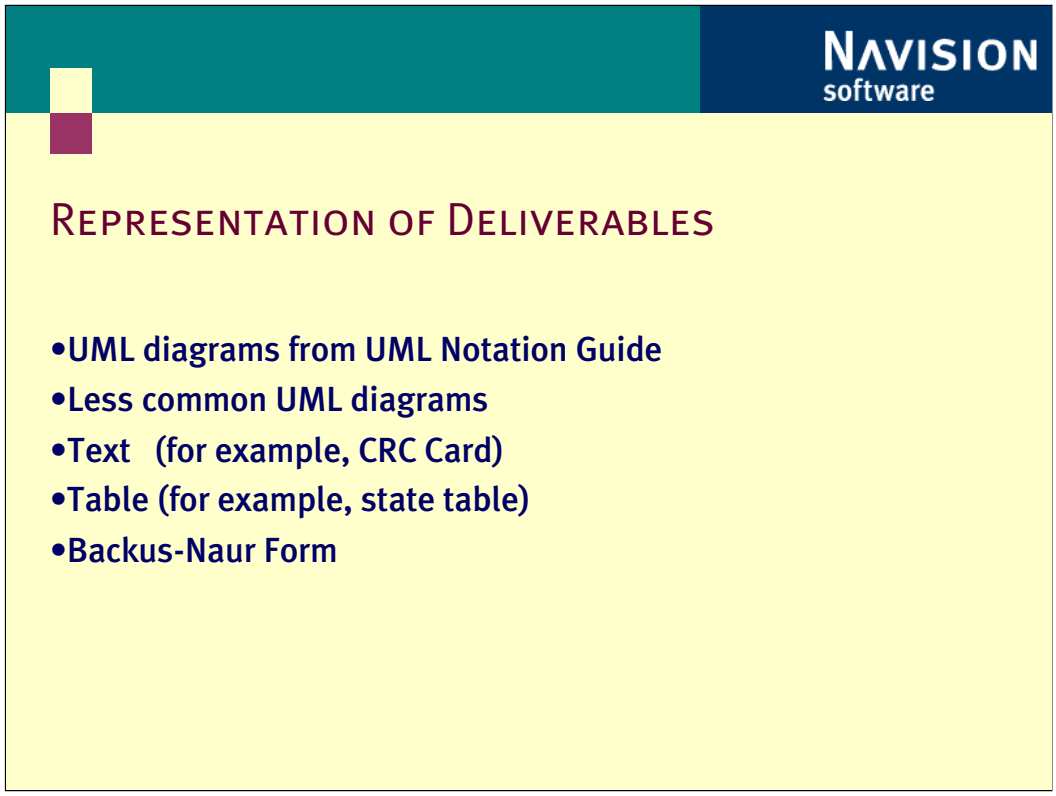
The dependency with the stereotype «trace» indicates that test cases can be based on use cases.

NAVISION software

Other Applications of the Pattern: On-line Documentation



The pattern can be used for designing online user documentation. Deliverables for 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.



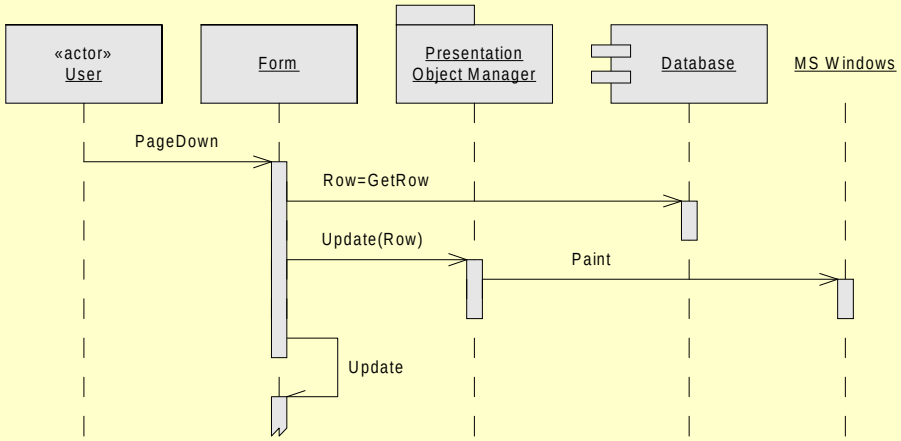
Design deliverables do not necessarily have to be described by UML. Practical alternatives to UML are Backus-Naur form (BNF), tables and text. The choice of the representation depends on the problem being described, as well as other circumstances such as who the intended reader is.

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.

NAVISION software

Less Common UML Diagrams: Interactions between Components

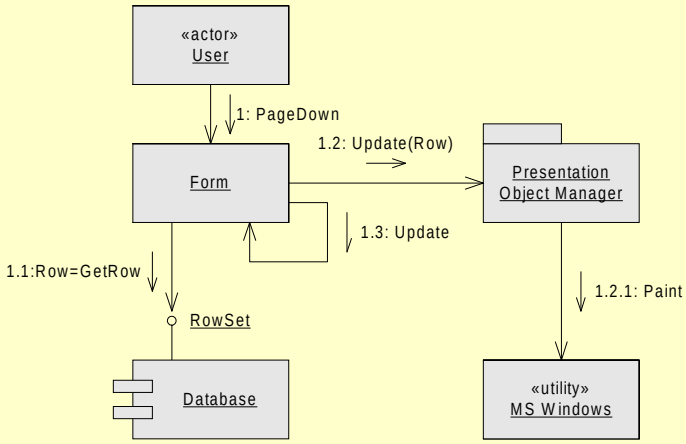


Interaction diagrams for subsystem, component and node interactions are sequence and collaboration diagrams in which classifiers are subsystem, component and node. These diagrams represent interactions between subsystem, component and node instances, without it being necessary to specify actual objects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

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 the slide.

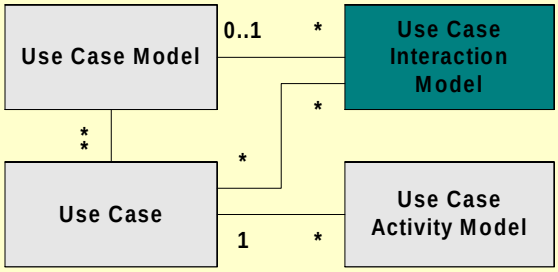
NAVISION software

Less Common UML Diagrams: Interactions between Components



NAVISION software

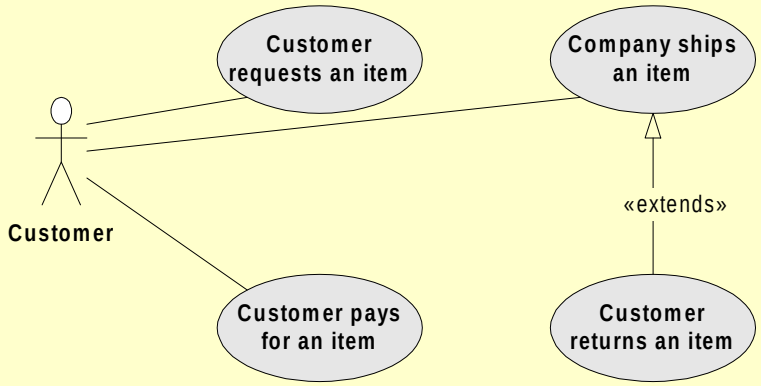
Less Common UML Diagrams: Use Case Interaction Model



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.

NAVISION software

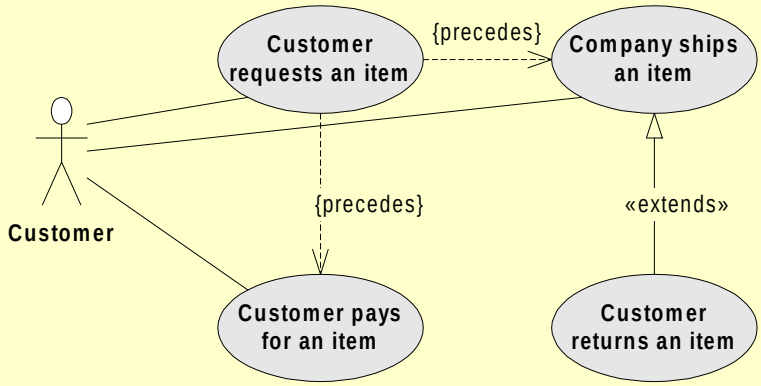
Allowable order of Use Cases



UML 1.1 cannot easily express that a customer first requests an item, then a company ships an item, and then the customer pays for an item.

NAVISION software

Allowable order of Use Cases: Use Case Diagram

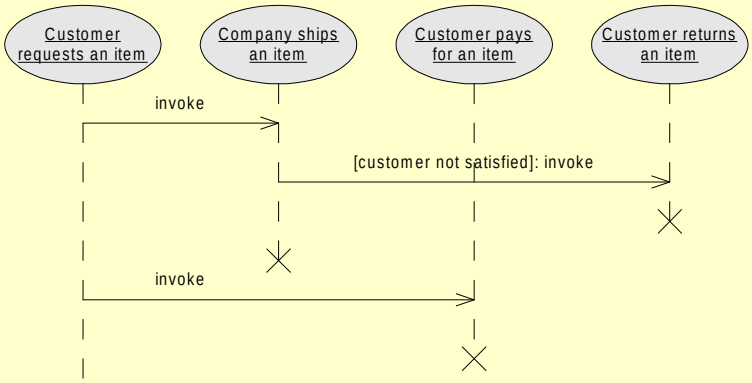


One of the solutions is to use constraints {precedes}, or dependencies «precedes» between use cases. Similar relationships exist in OML (OPEN modeling language), please see http://www.csse.swin.edu.au/cotar/OPEN/OPEN.html.

However, this is still a static structure diagram, not a scenario!

NAVISION software

ALLOWABLE ORDER OF USE CASES: SEQUENCE DIAGRAM



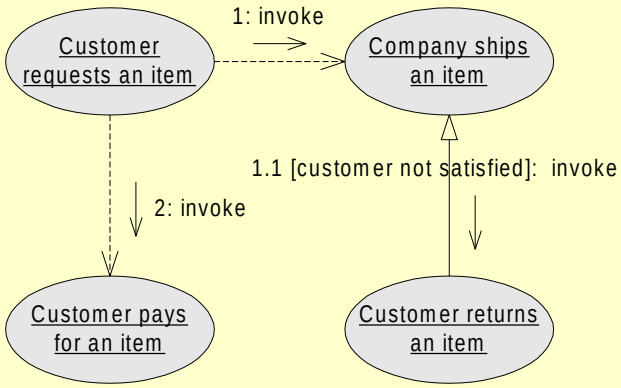
The *use case interaction model* specifies typical sequences of use case instances. In contrast to object, component and node interaction models, where a scenario is described as a sequence of messages, the use case interaction model describes the scenario as a sequence of use cases. This model is the only UML deliverable that can describe a scenario consisting of other scenarios.

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* 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 use case activity diagrams in which action states map to subordinate use cases.

NAVISION software

ALLOWABLE ORDER OF USE CASES: COLLABORATION DIAGRAM

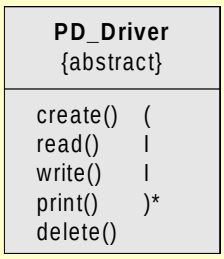


The collaboration diagram showing the same scenario.

NAVISION software

Interface Specification: Backus-Naur Form

Create; (read | write | print)*; delete;



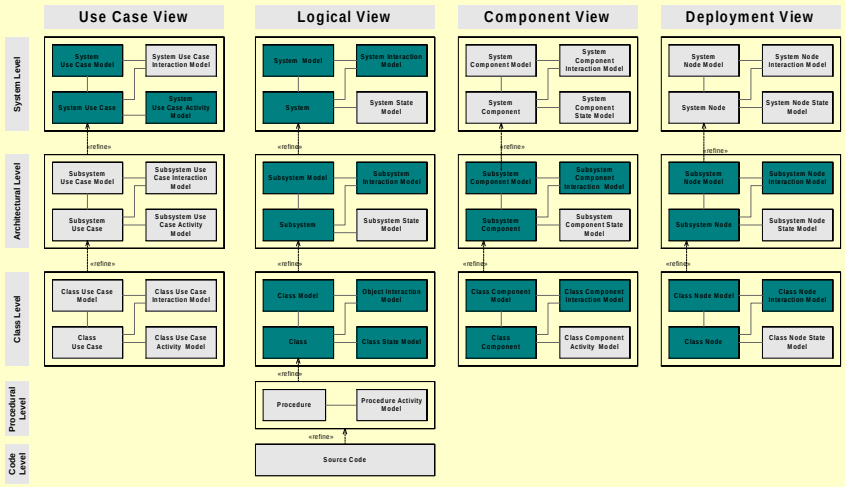
Backus-Naur form (BNF) represents scenarios with one or two participants or a valid order of operations of one classifier. Therefore, BNF is convenient for specifying interfaces.

The slide 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.

In simple cases, BNF expressions can be placed directly into the operation compartment of the class.

NAVISION software

OBJECTORY METHOD



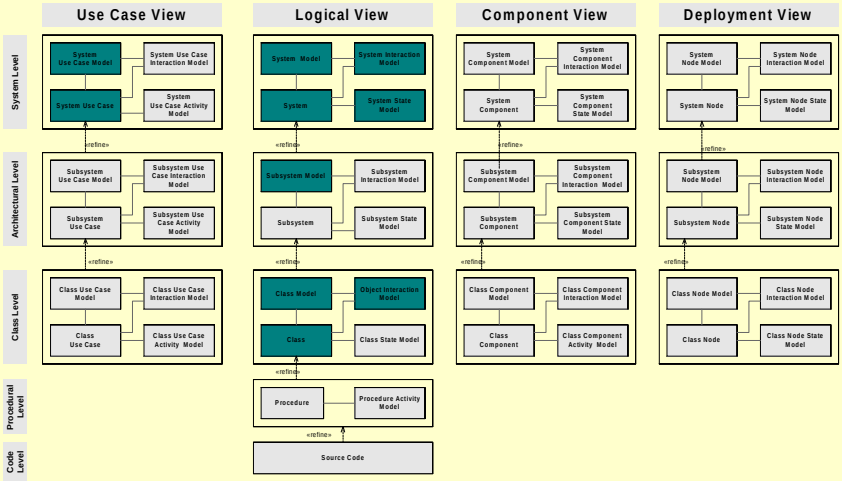
The deliverables of the Objectory method are structured on use case, logical, deployment, implementation and process views, and tier, architectural, and class levels.

Deployment and implementation views contain only component and node models and component responsibilities. The 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 the use cases (in other words, according to their collaborations with external actors).

NAVISION software

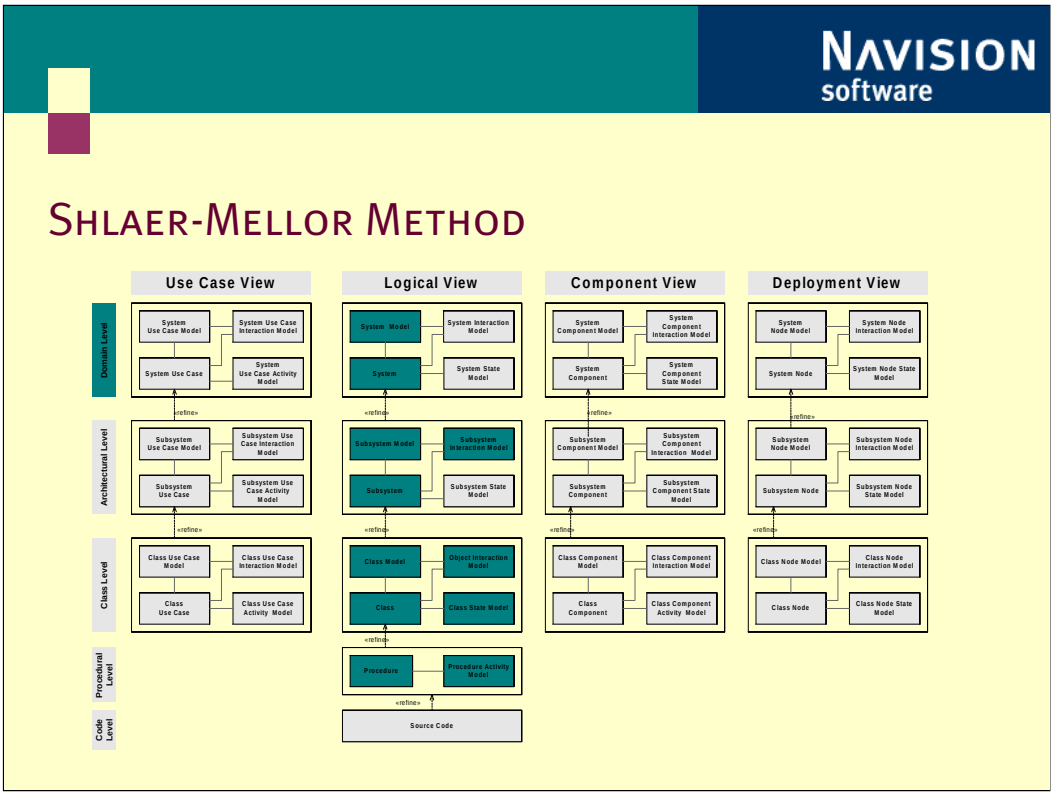
FUSION METHOD



The Fusion method is a method with a succinct and consistent system of deliverables that is orthogonal, which means that one fact about the product is stated only in one place.

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). Deliverables are structured according to the refinement between levels of abstraction.

The new Fusion Engineering process (also known as Team Fusion) produce also use cases and use case model.



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 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 procedural 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 the pattern of four deliverables.

NAVISION software

More Information

Pavel Hruby
e-mail: ph@navision.com
Internet: www.navision.com/services/methodology

If you have other solutions or comments please let me know.

Whether you have downloaded the presentation from the web site, or obtained it from some other source, I am interested in hearing your opinion or alternative view.