The four faces of UML

(Version of May 18, 1998)

Trygve Reenskaug

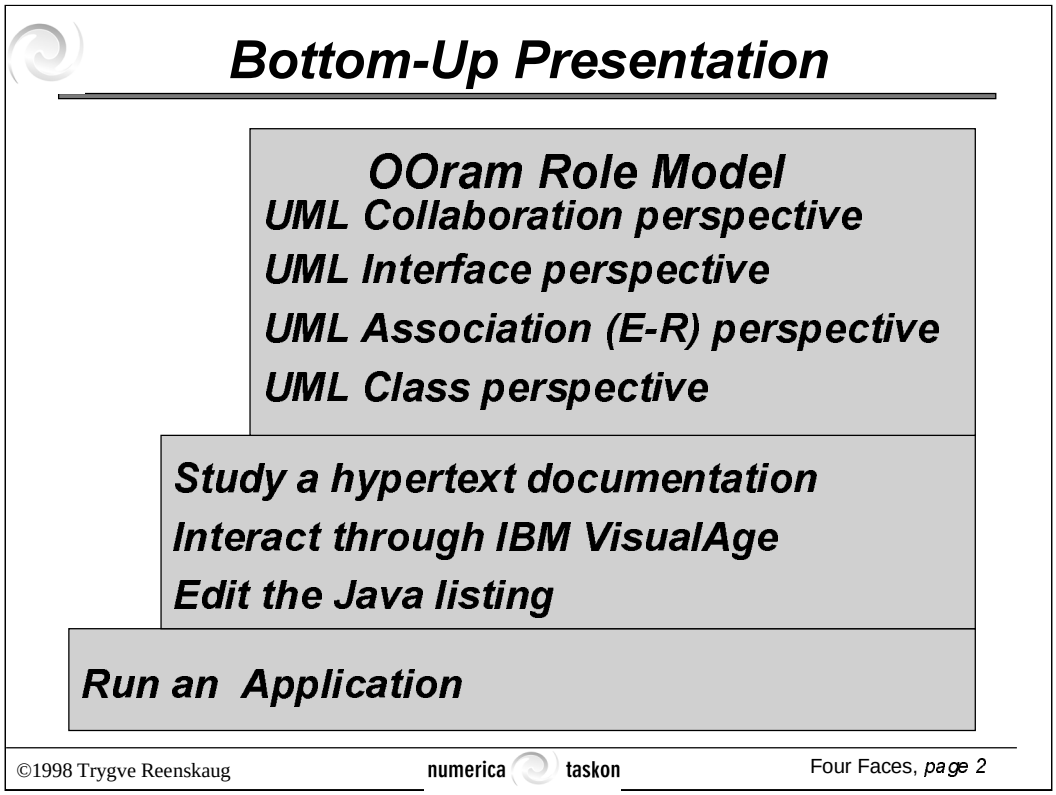
UML offers 4 different perspectives useful for different purposes.

- ***Do we need them?***
- ***What do they highlight and hide?***
- ***So what?***

UML offers a varied and powerful language for describing object oriented systems.

We like to view the UML perspectives as belonging on four different axis.

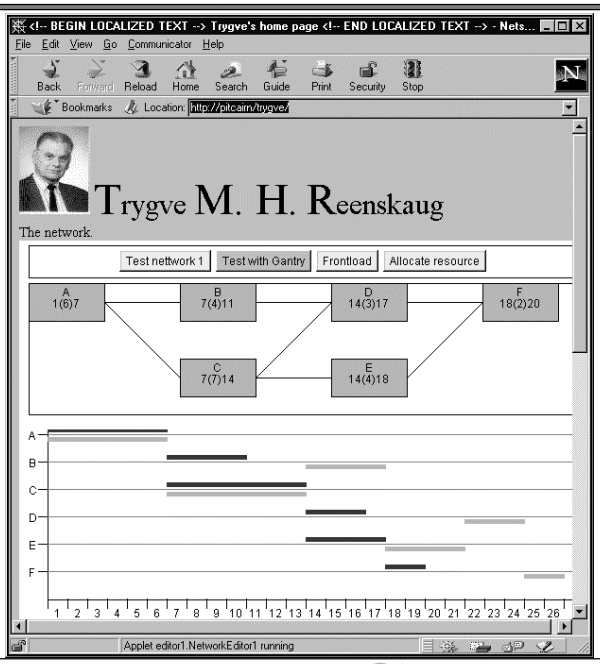
We will explore these perspectives: What are they and why do we need them?



We will discuss UML in the context of a concrete example

- Run the example
- Look at the code. Programming environment important because it is ONE representation.
- Add modeling if above is insufficient. Select modeling concepts that highlight problem areas that are still unclear.

Scheduling with Default Resource



Model
Code
Run an Application

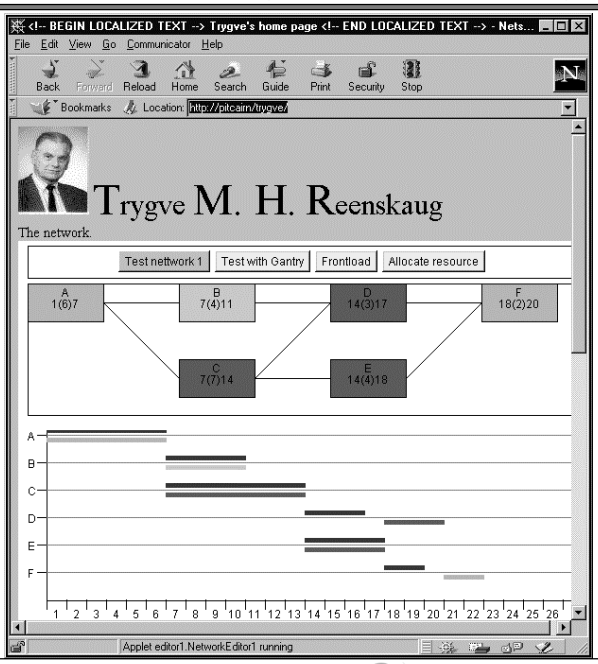
numerica taskon

This toy sample user interface is a Java Applet that is run from a web browser. The interface interacts with a background service that is also written in Java.

The top bars in the bar chart shows the earliest times when the activities can be performed. The bottom bars show when the activities all have to be done by the same resource, and this resource can only serve one activity at the time.

©1998 Trygve Reenskaug
numerica taskon
Four Faces, *page 3*

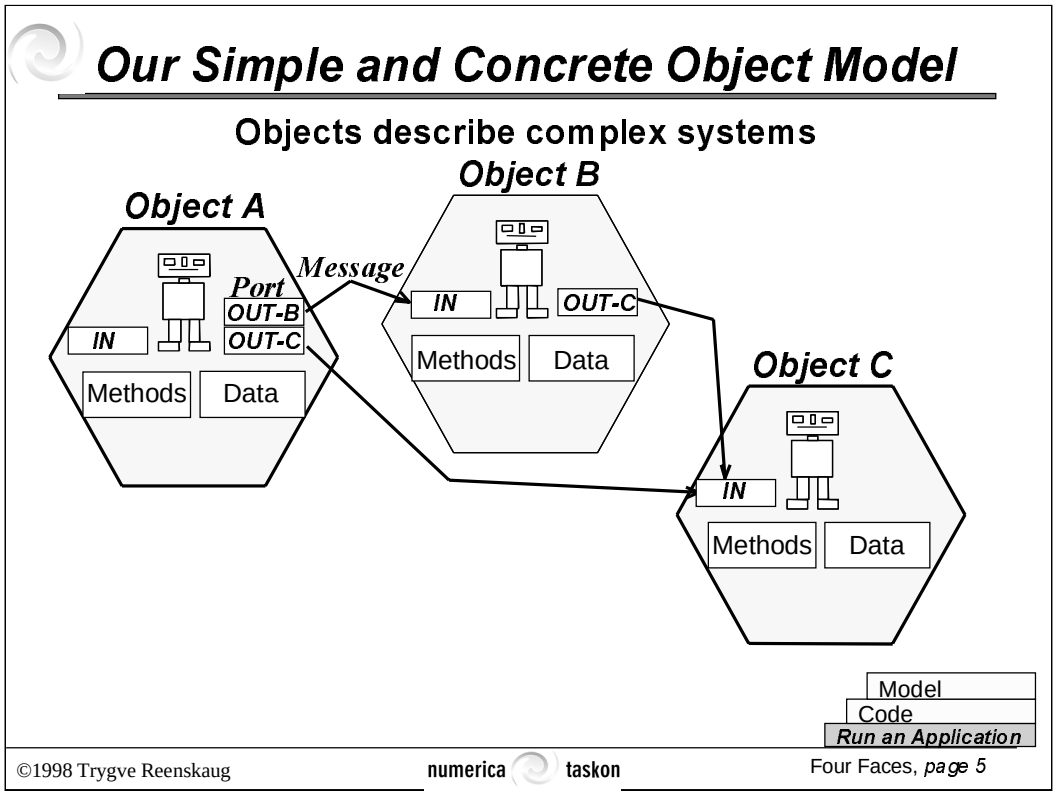
Scheduling with Gantry Crane



Model
Code
Run an Application

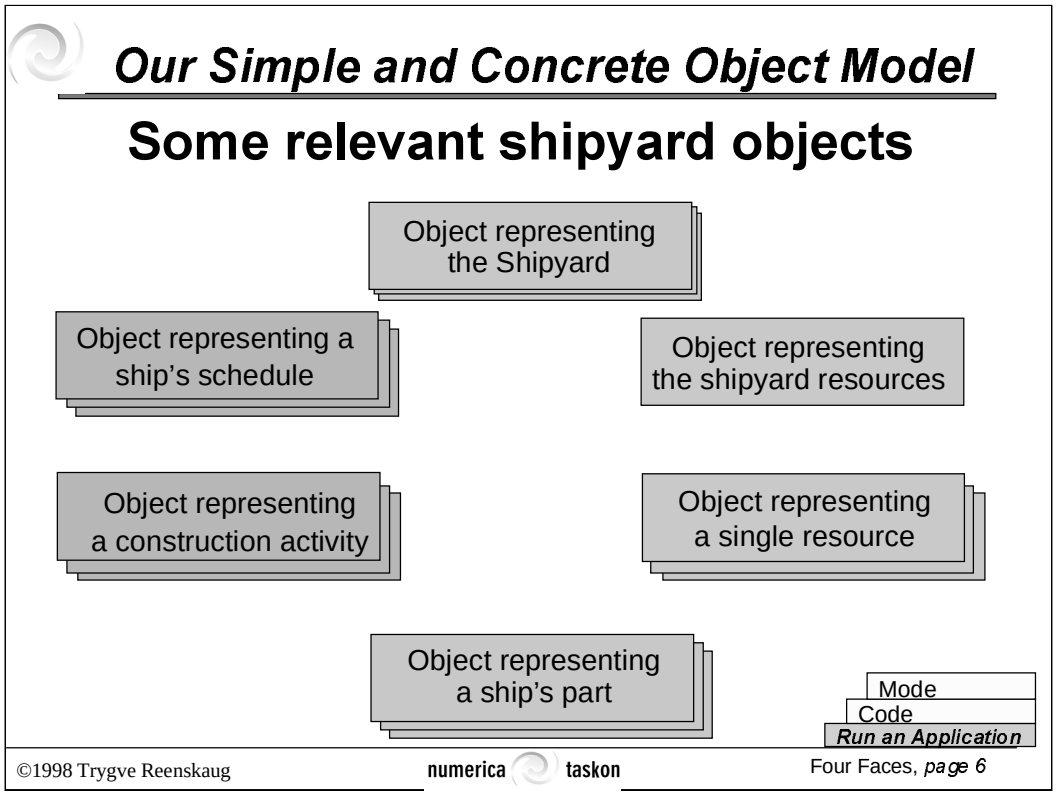
An example of specialization: Some activities use a Gantry crane. This crane has two independent hooks: One can lift a light load and the other can lift any load.

Activity B is to lift a light load. It can be performed concurrently with activity C, which is a heavy load. Activities D and E are both heavy loads, they have to be performed in sequence. Activities A and F use the old resource that can serve one at the time.

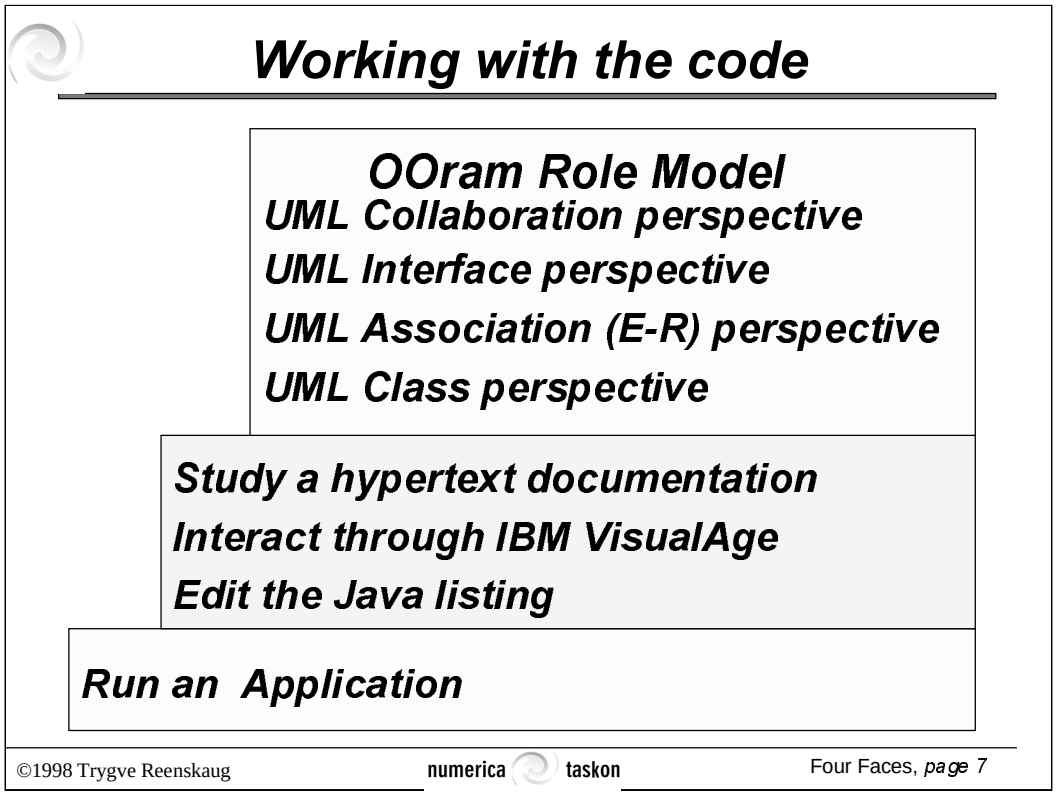


Traditional activity planning systems are programmed with a centralized logic working on standardized activity records. The current demo replaces activity records with objects so that different activities and resources can behave differently according to their nature.

Thinking about this activity scheduling system implies thinking about interacting objects. What is our strategy for performing the scheduling. What are the objects and what are their responsibilities?



Our example is taken from a shipyard. We represent important aspects by objects and let scheduling be performed through negotiation between these objects.



Given this solution: In which ways can we create and study the underlying program?

An important goal for all programming languages is to bridge the gap between the programmer's understanding of the solution and the actual code.

We will here look at three contexts for presenting the Java code of our example.

Part of Activity *Java source*

```
/** Activity frontloading operation
 *   Reports earliest completion of a predecessor
 */
public void frontload (int time ) {
      earlyStart = Math.max (earlyStart, time);
      if ((counter += 1) >= predecessors.length) {
            int fin = getEarlyFinish();
            for (int i=0; i<successors.length; i++) {
                  successors[i].frontload(fin);
            }
      }
}
```

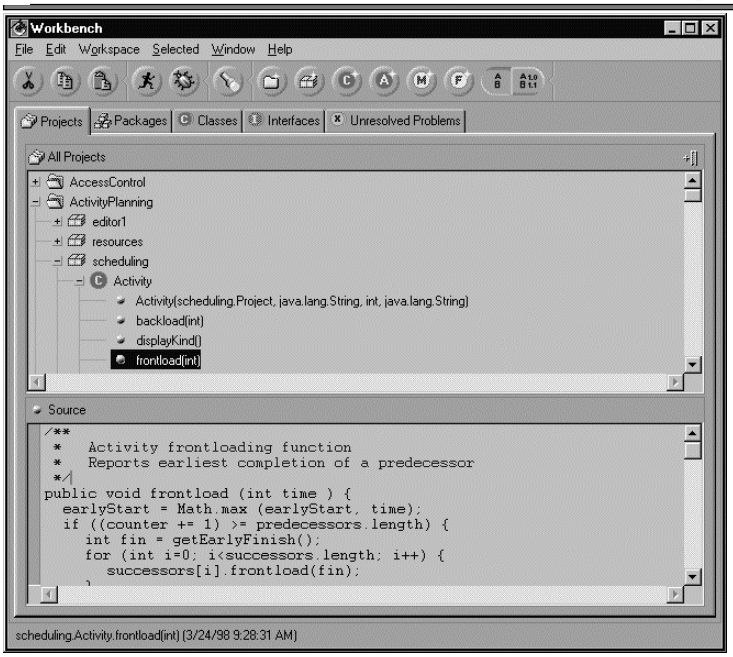
~13 classes ~102 methods ~1500 lines

Model
Edit/Java listing
Run an Application

The simplest is to use straight text files and a text editor such as EMACS. In Java, convention is that each class is stored in its own file. We use the directory structure of the file system to manage concepts such as projects, packages and class libraries.

©1998 Trygve Reenskaug
numerica taskon
Four Faces, *page 8*

IBM VisualAge® Workspace



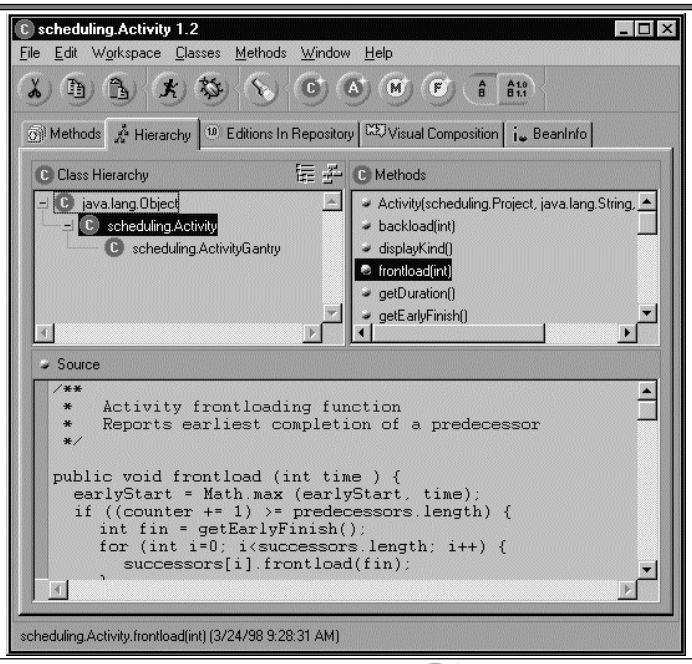
Model
Edit/VisualAge
Run an Application

numerica taskon

IBM VisualAge for Java is an example of a higher-level environment for writing and exploring Java code. The environment manages the code and can be considered to be an extension of the programming language.

This view is a browser for the tree consisting of projects, packages, classes and methods.

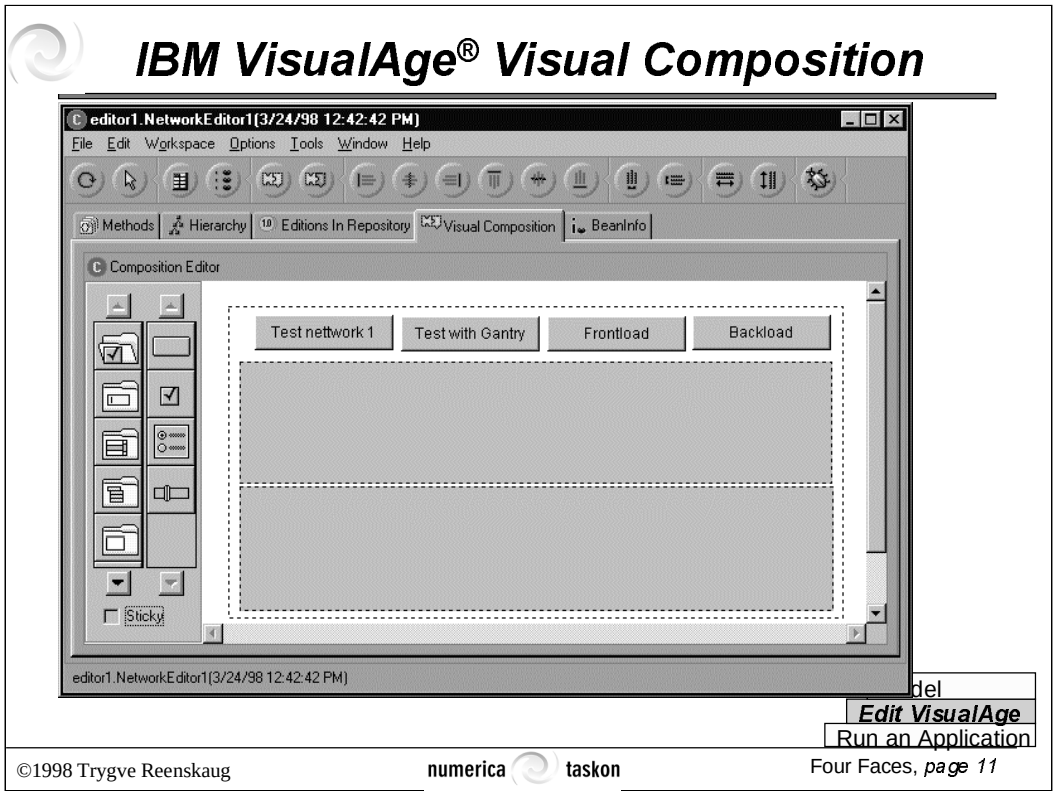
IBM VisualAge® Class Hierarchy



Mode
Edit VisualAge
Run an Application

numerica taskon

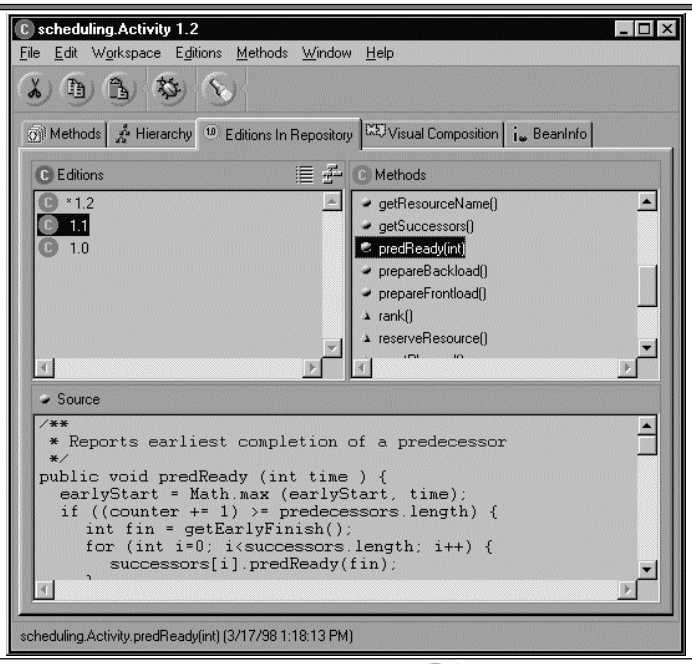
Here is another perspective on the code, showing the class hierarchy and the methods that will be parts of all instances. Filters permit flattening of the code so that *all* methods can be visible even if implemented in a superclass.



Here is a powerful code generator for user interfaces. The programmer paints the appearance of the interface, adding buttons, text fields, menus, etc. (VisualAge has a capability for defining actions. (Not illustrated here).

Many (most) programming environments provide something similar to this.

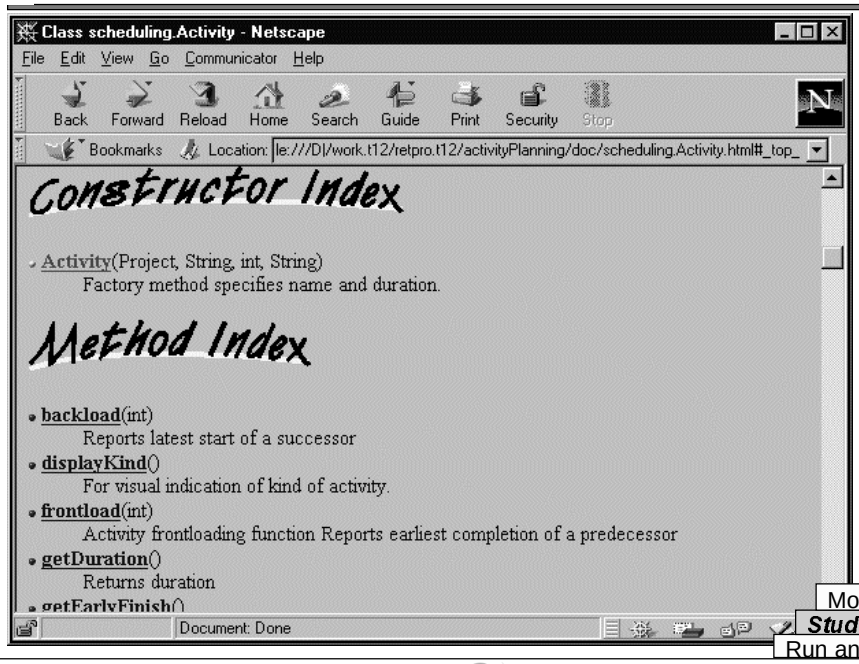
IBM VisualAge® Version Management



Model class
Edit VisualAge
Run an Application

Another extension of the programming environment is a version management system that appears as a seamless extension of the programming language.

Hypertext document: Activity class



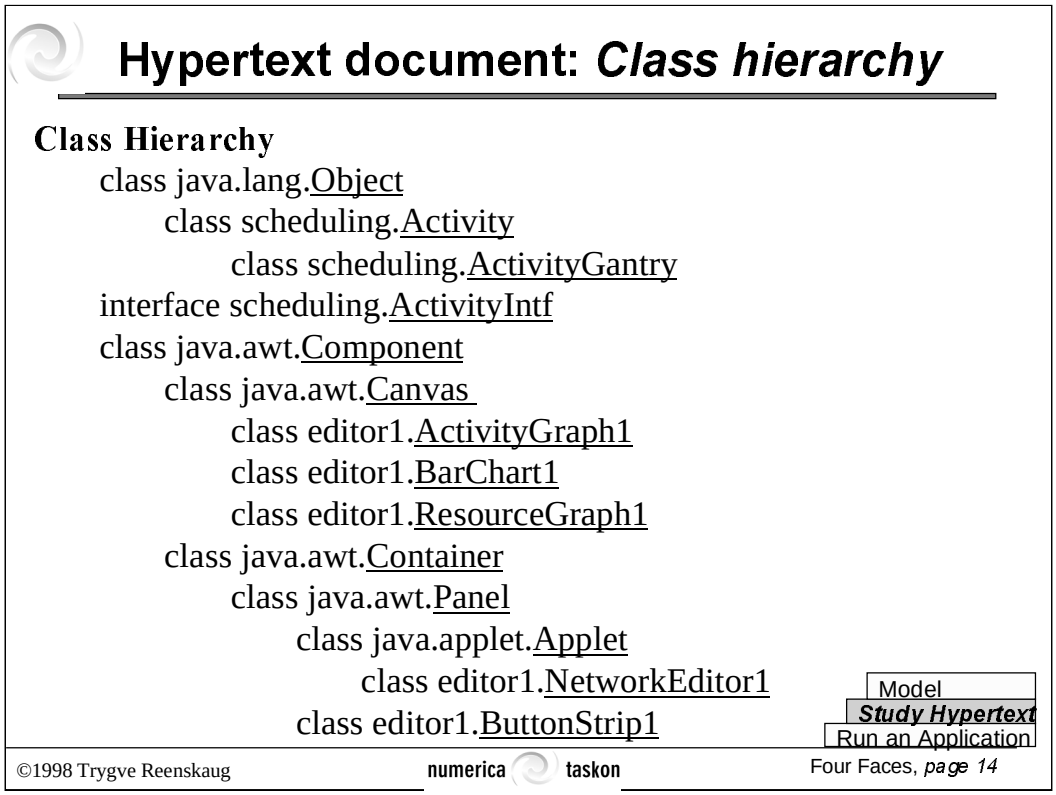
Model class
Study Hypertext
Run an Application

numerica taskon

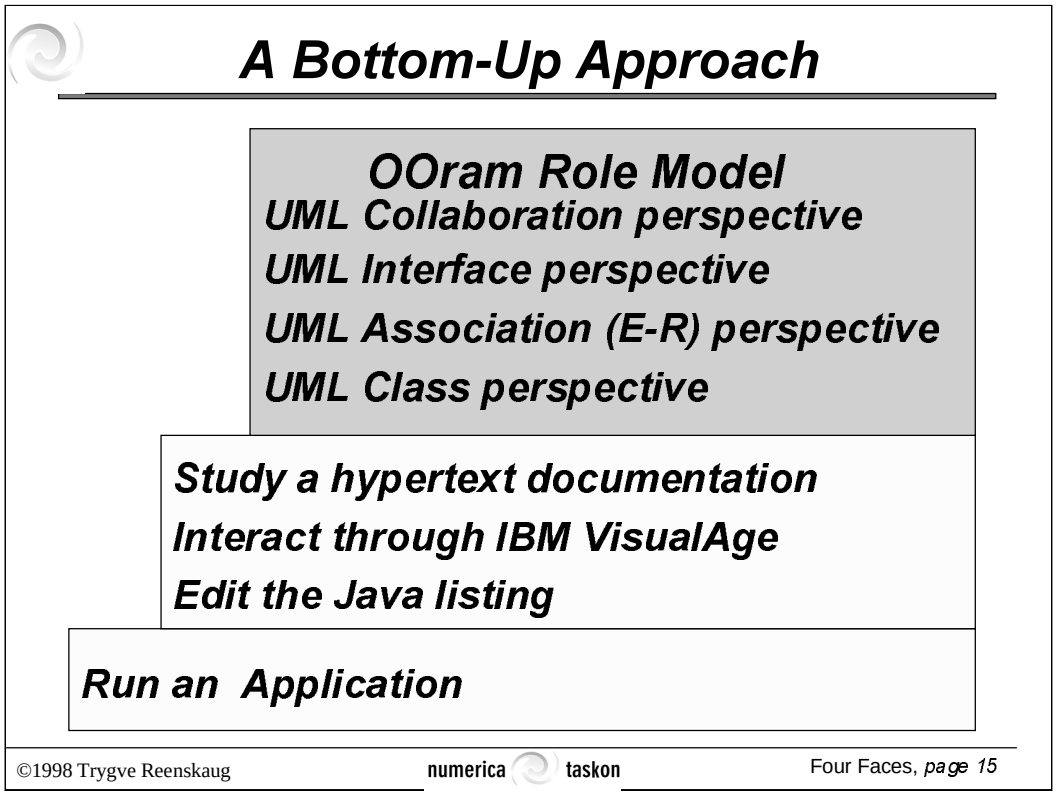
Comments are in Java enclosed by a special construct: */* comment */*. By convention this variant is defined to be part of the documentation: */** documentation comment */*

The programming environment uses this information to produce a richly connected hypertext document. This slide shows a small part.

We have now left the programming environment; we cannot edit and compile from the hypertext version. But the document is a *view* of the code since it is produced completely automatically from the code.



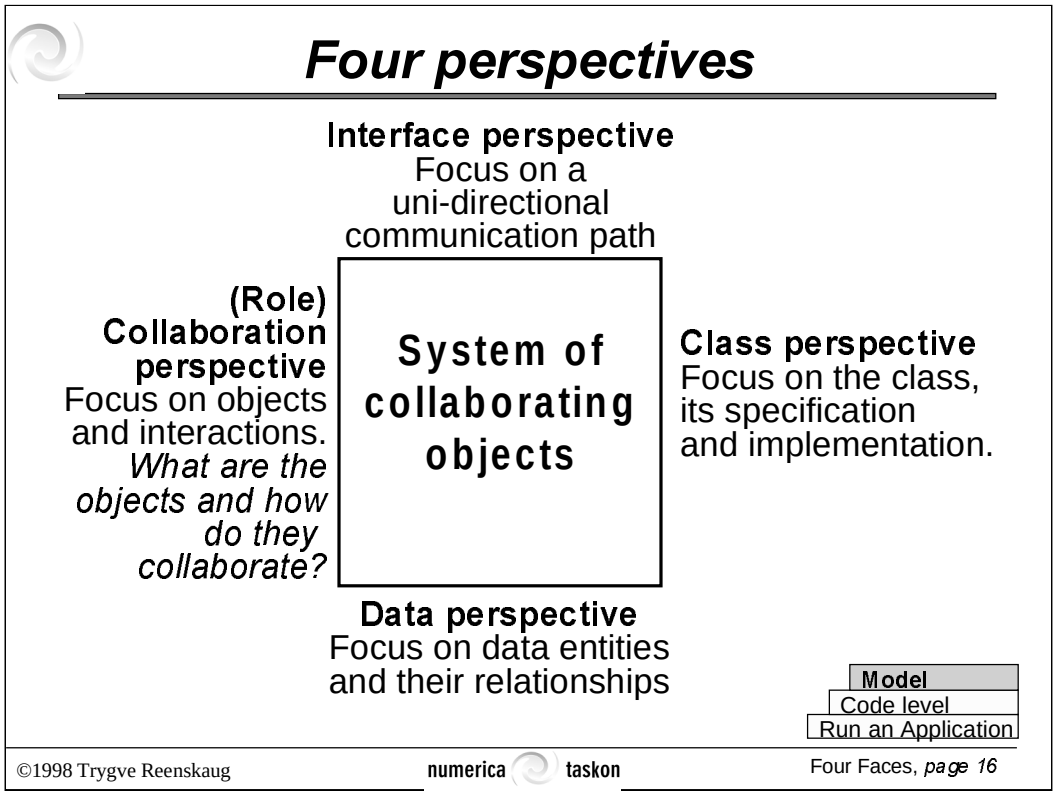
This is another of the automatically produced hypertext reports, it shows the class inheritance hierarchy and can be used to navigate in the documentation.



Given what we can get from the code and the programming environment, we now finally come to UML. What does it give us that we haven't got already and do we need it?

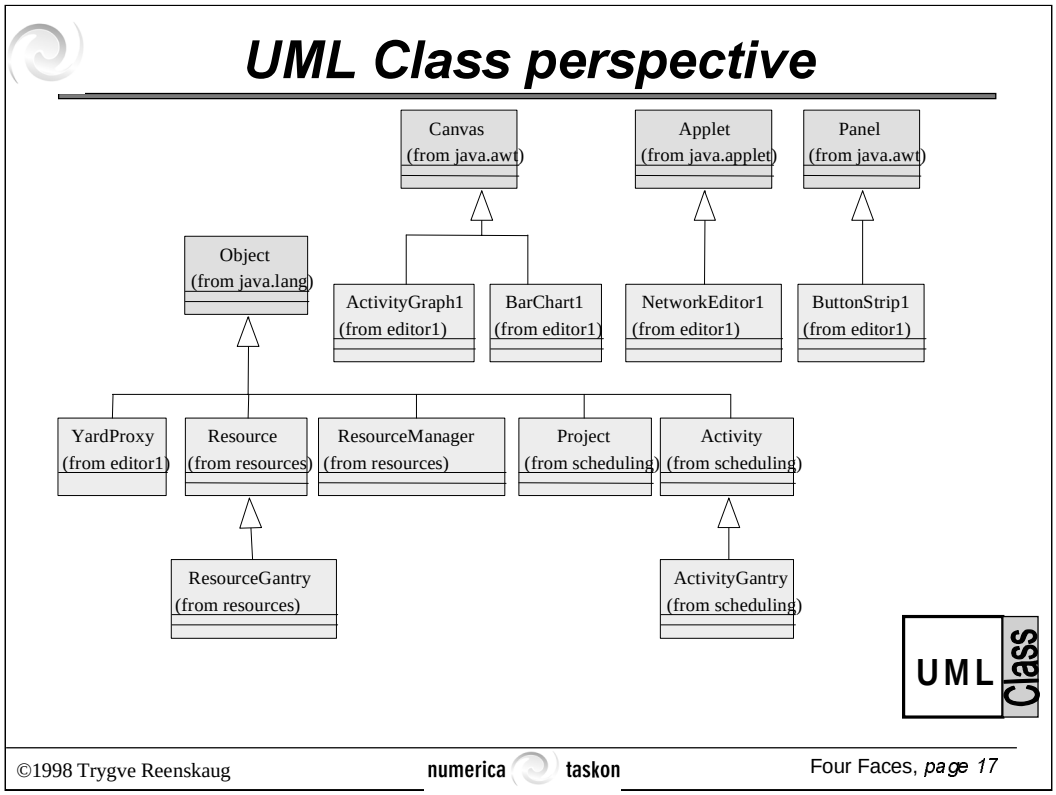
UML's offerings have better be important to us, because we now need to create and maintain a separate description outside the program itself.

Clearly, the weaker our programming environment, the more important will be the separate program models.



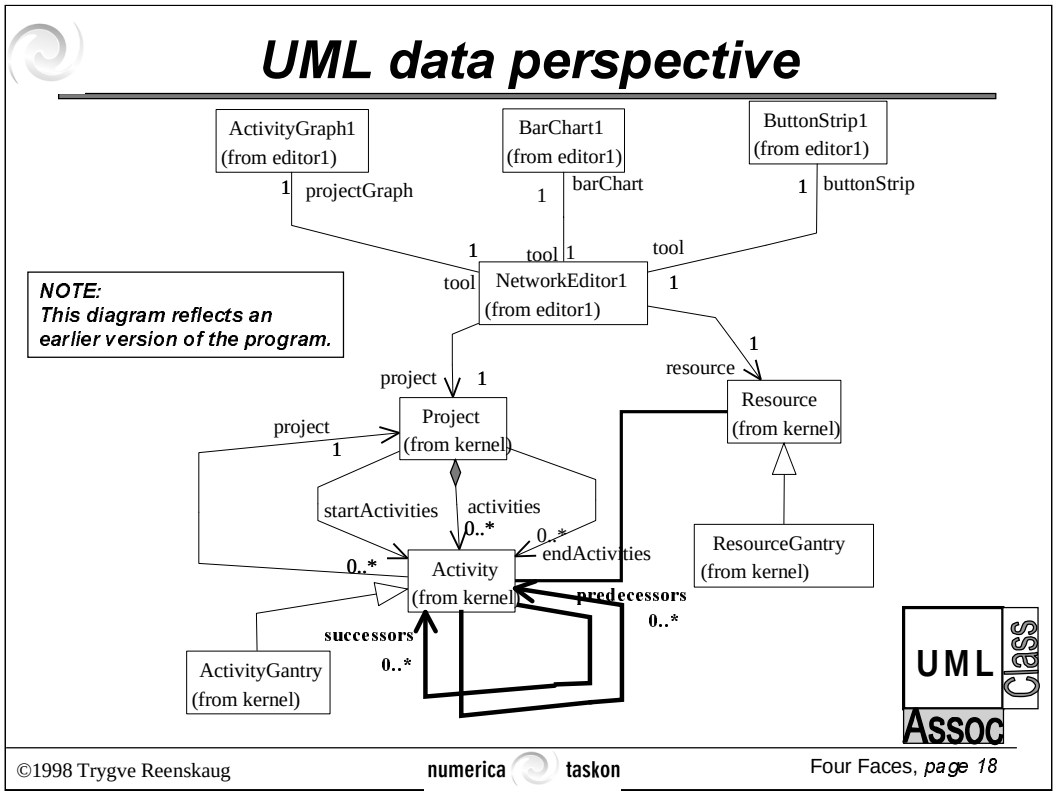
UML is a very rich modeling language, encompassing most important object oriented modeling paradigms that have been proposed over the past years.

We will focus on the four important perspectives that are illustrated in the slide.



The class perspective shows the classes and their inheritance relationships.

The class perspective can also show class attributes and method interfaces. We have not illustrated this capability in this slide because of space limitations.

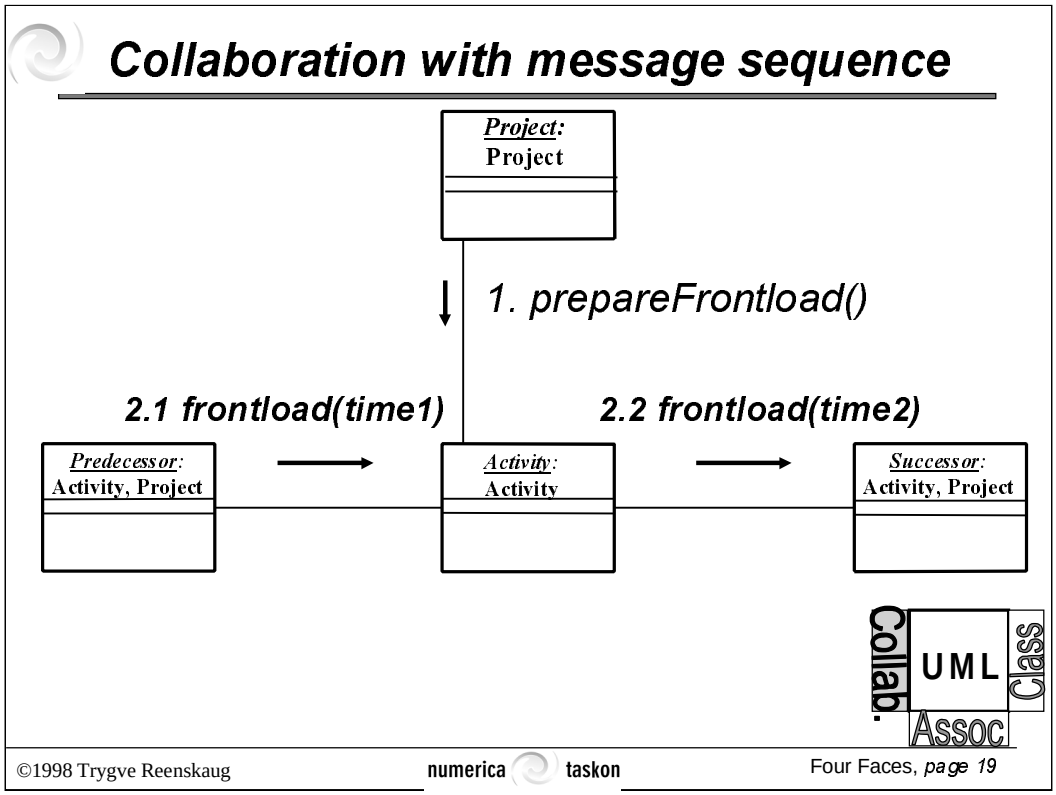


A UML association shows a relationship between classes. This is exactly analogous to the Entity-Relationship modeling used for modeling relational databases. As in E-R modeling, each class represents the set of all instances of that class.

The logic of this diagram was generated automatically from the Java code with the Rational Rose® reverse engineering tool. An exception is that Rose did not find the *aggregation* modifier that says that Acitivities are aggregated in the Project.

A more important exception is that Rose did not find the association between Activity and Resource. In the program, this essential association is implemented indirectly; the Activity knows the name of the resource and asks the Project for its reference whenever it needs it. (It can of course be added manually as we have done here. In fact, there is no strict relationship between UML *associations* and Java program *references*).

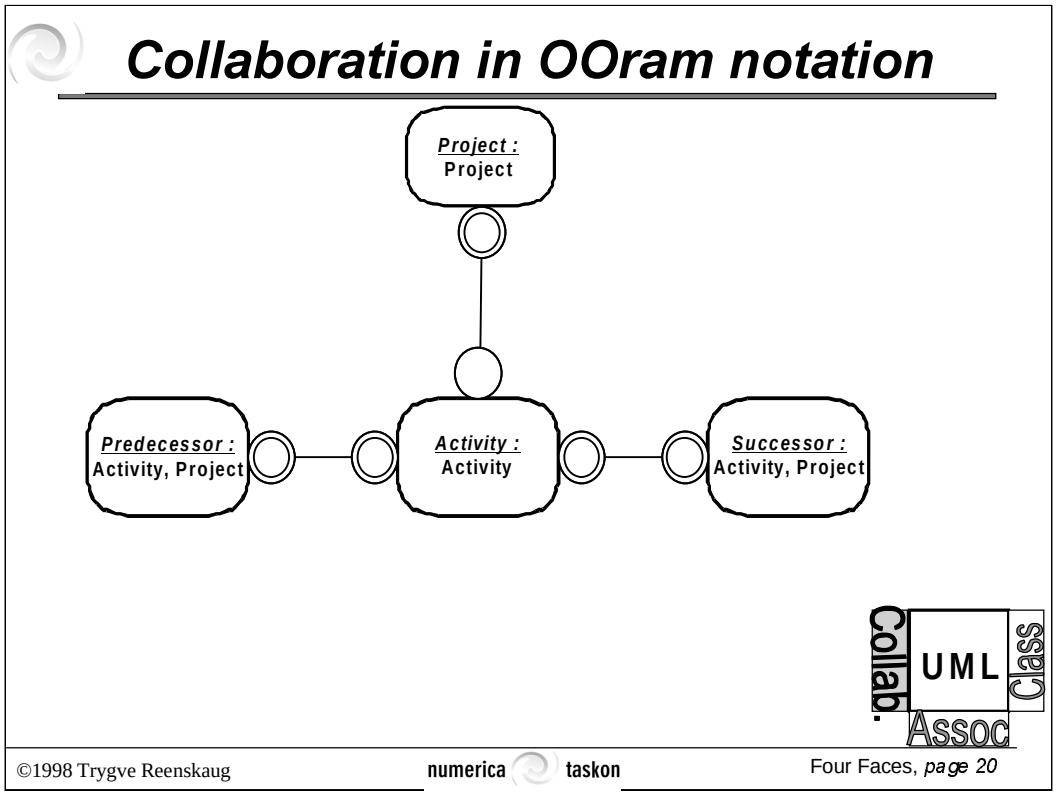
Notice that the Activity class is associated with itself through the *successors* and *predecessors* associations. These associations do not identify the predecessor and successor objects, but merely state that an Activity object is associated with some other Activity objects. Specifically, the diagram does not say that the successors of one Activity objects count this object among their predecessors.



We change our perspective from classes to objects; from sets to individuals. This permits us to study patterns of interacting objects: What are the objects, what are their responsibilities, and how do their collaborate to achieve the system functionality.

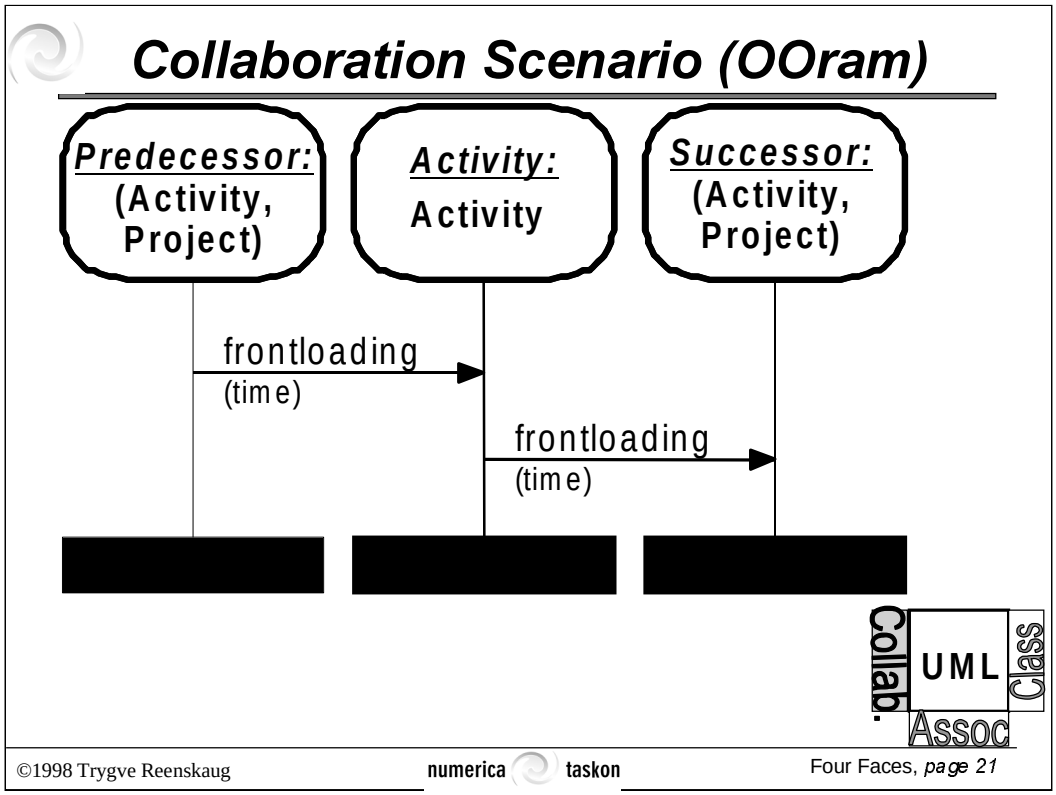
The (classifier) *role* represents the object's position in the structure. We see that instances of the Activity class play different roles: *Predecessor*, *Activity*, and *Successor*. Instances of the *Project* class are also *Predecessors*, in this way a *Project* can start the frontloading operation. The *Project* also plays the *Successor* role so that it can receive the final scheduling messages.

We achieve a *separation of concern;* a collaboration describes the objects involved in one or more functions, focusing on the objects involved and describing the relevant object properties.



An important UML extension mechanism is the *stereotype* which adds more semantics to the basic modeling elements. A stereotype may introduce additional values, additional constraints and a new graphical representation.

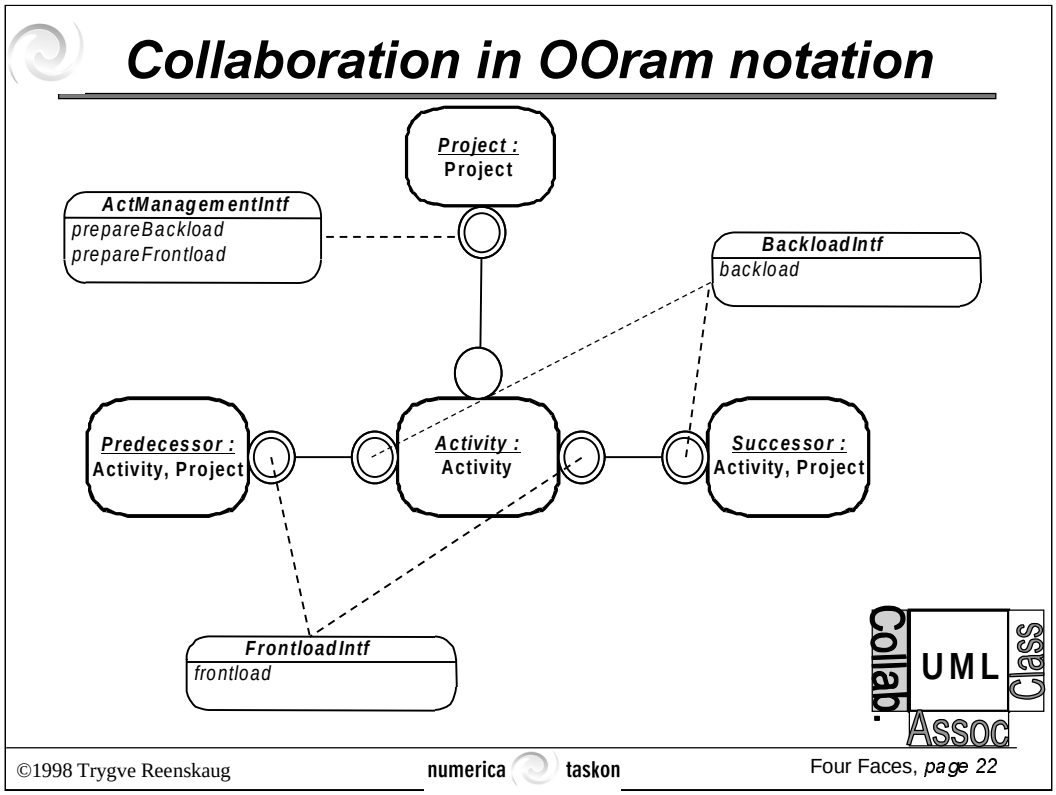
We here illustrate the OOram extension to the UML collaboration. The semantics of the OOram role model is more precisely defined than the UML collaboration. The most important practical consequences are *separation of concern* and model refinements through *synthesis* (to be discussed later).



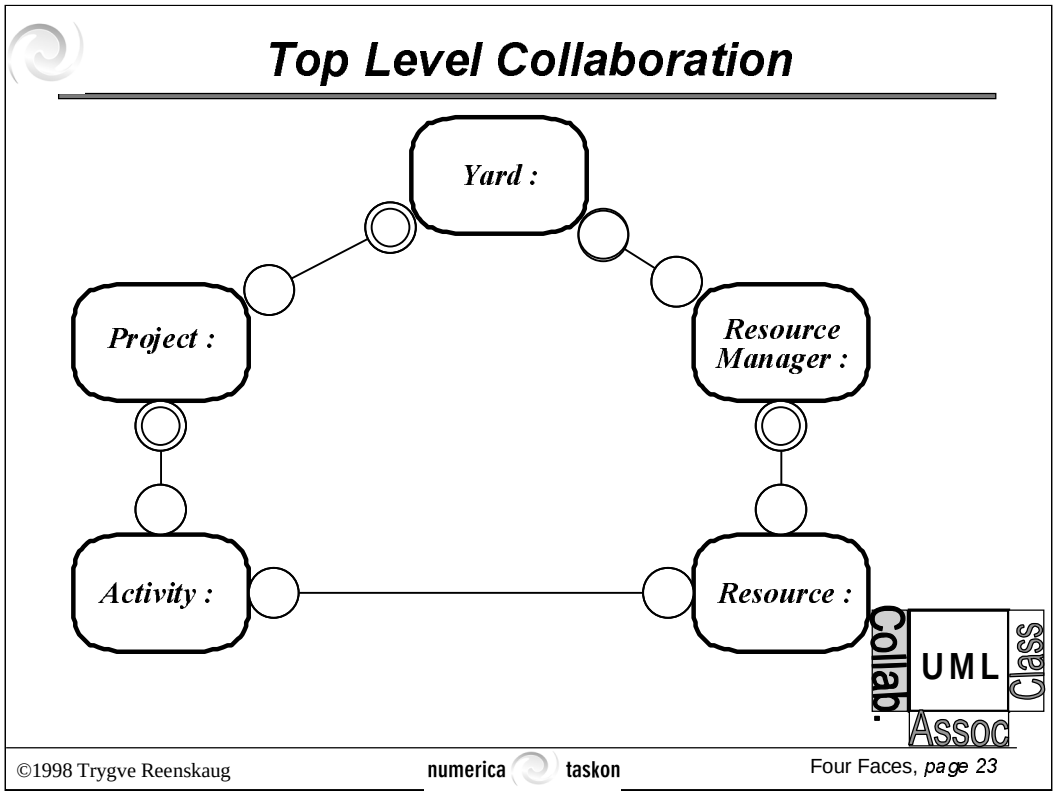
A collaboration (role model) represents a pattern of objects that interact to perform a given *use case* in order to reach a specified goal.

The scenario is one way of describing the dynamics of interacting objects.

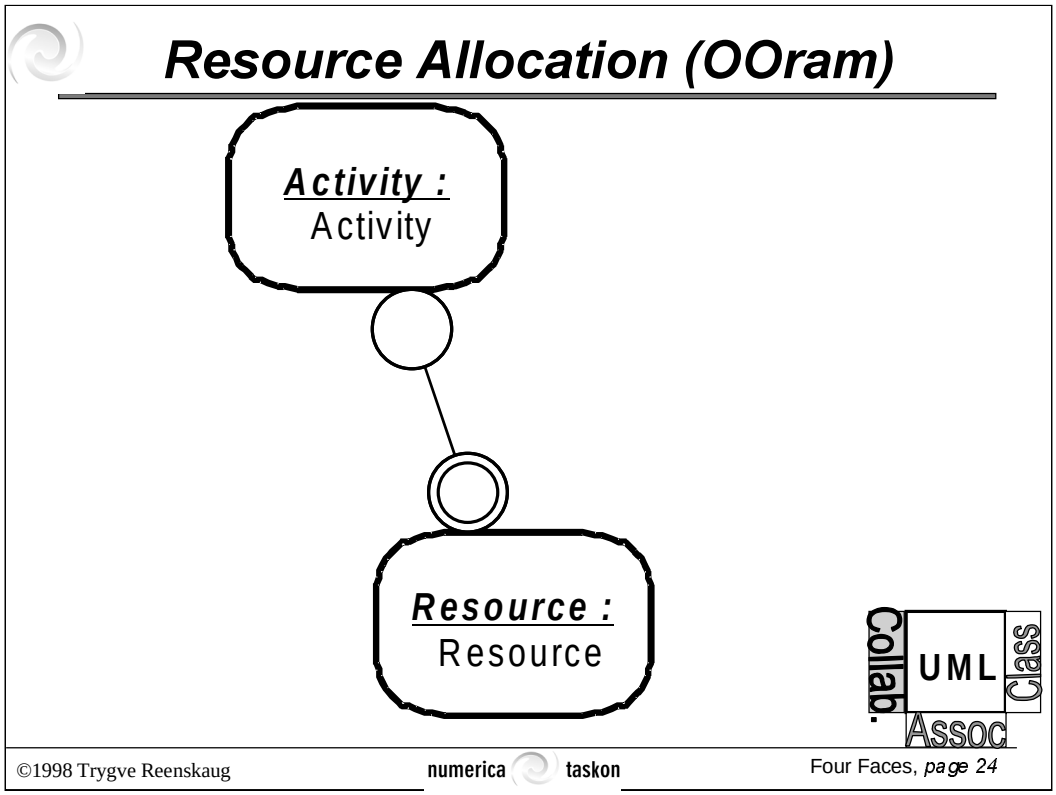
Note that the nodes of a scenario cannot be classes, because the classes hide object identity and we cannot know which object sends and which object receives the messages.



We see the ClassifierRoles as super-ellipses and the AssociationEnds as small circles, called *ports*. A port represents an object's *reference* to a collaborator, it is implemented as a variable or a function or some kind of constant. Java permits the typing of references with *interfaces,* we have illustrated some interfaces in this slide.

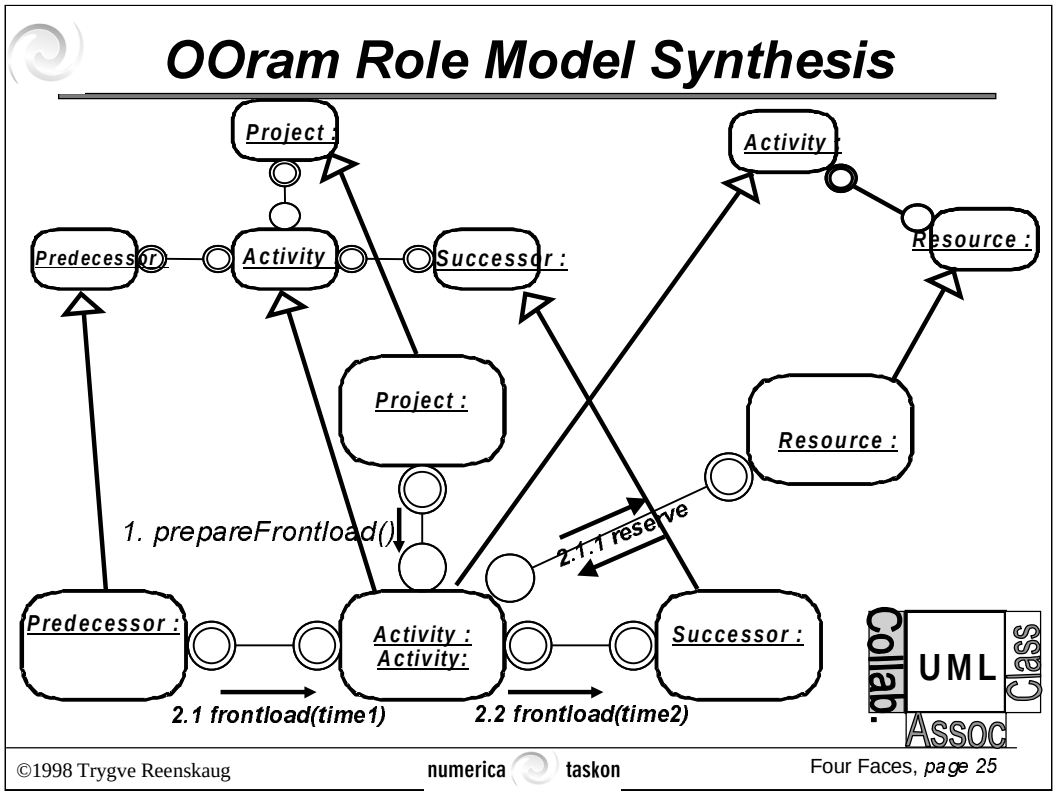


This is an illustration of the OOram *separation of concern.* We have previously studied the details of the scheduling operation. We now take a wider view and model the overall system architecture. Compare to figure 6. We see that the current version of the program reflects our initial intuition fairly well. The exception is that we do not model the product; this has been postponed to a future version.



We again change Area Of Concern, this time to model the negotiation between an Activity and its Resource for allocating access.

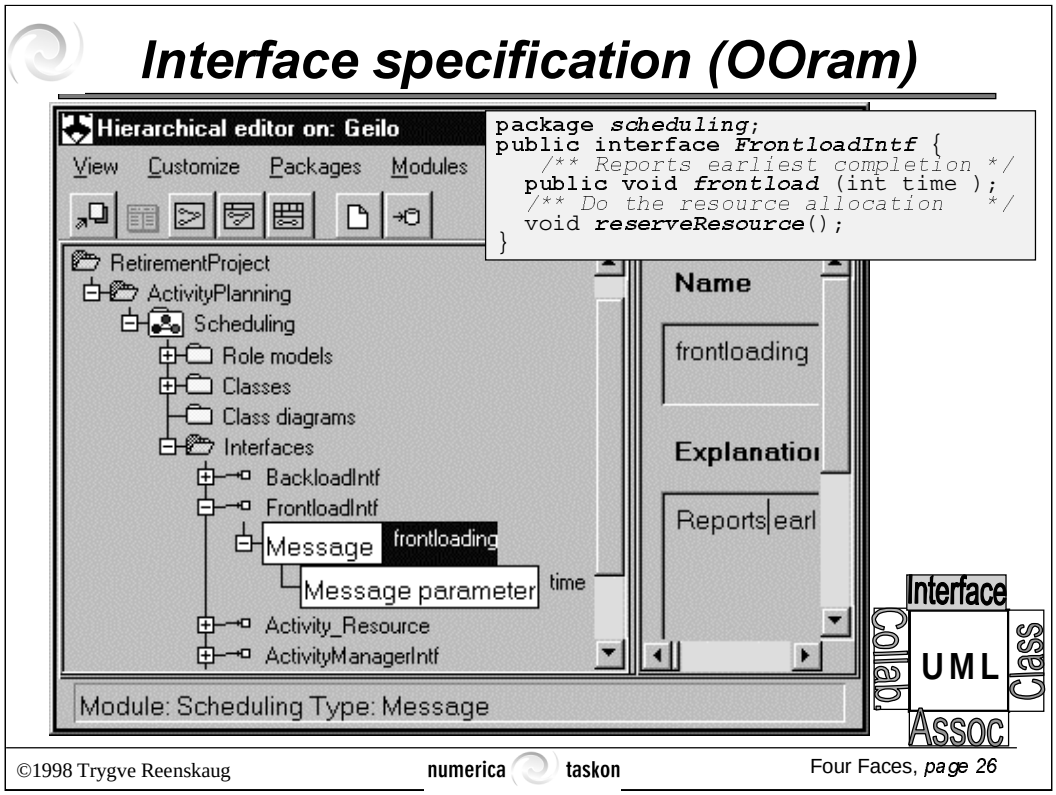
We do not give the details here, but note that we model and program each Activity-Resource pair to reflect their special characteristics. The Gantry craneResource can, for example, ask the lifting Activity for the weight of the load before it makes the appropriate allocation.



We now use *synthesis* to create a composite model that shows the scheduling process combined with the resource allocation process.

The resulting *derived model* is linked to two *base models* and reflect the static and dynamic properties of these models. So the use cases of the base models are all included in the derived model, either as explicit use cases or as subordinate use cases that become part of other functions.

Technically, every role in every base model is mapped onto some role in the derived model. The ports (association ends) of the derived model includes the combined interfaces of all the corresponding base ports. And the behavior of the derived model as described by its scenarios includes all the behaviors of the base models.



We finally show the *interface perspective*. Interfaces can be used to define the external properties of a class. In collaboration modeling we are using them to specify the messages that may flow along the different collaboration paths.

We here show the OOram and Java perspectives on the same interface. The OOram interface can be specified informally or formally according to what is most convenient.

So what?

- **Select *your* programming environment that suits *your* needs.**
- Use UML class perspective to design class hierarchies
- UML data (E-R) perspective may be superseded by collaborations
- **Use interface modeling for distribution and reusable components.**
- **Use UML collaborations & OOram role models when working with interacting objects**
- **Use OOram synthesis for reusable patterns**

Our advice must necessarily be subjective, based on our own preferences and experience.

We find that a powerful programming environment covers most of our needs. Its main weakness is that it neither isolates the different functions (use cases, subjects), nor shows how they are accomplished in the object structure. We therefore add collaborations (role models) for designing and documenting critical functions.

On a more advanced level, OOram synthesis is very powerful for designing, documenting and applying reusable patterns and frameworks.

References

- http://www.ifi.uio.no/~trygve
  http://www.numerica-taskon.no
- Reenskaug, Wold, Lehne: *Working With Objects.* Manning/Prentice Hall 1996. ISBN 0-13-452930-8
- *Unified Modeling Language (UML).* Proposal to the Object Management Group. Version 1.1, September 1, 1997. Documents can be retrieved from the OMG at ftp://ftp.omg.org/pub/docs/ad. Document numbers 97-08-02 through 97-08-05 define the standard.
- Egil P. Andersen: *Conceptual Modeling of Objects. A Role Modeling Approach.* Dr Scient theses. Dept. of Informatics, University of Oslo. 4 November 1997.
- Trygve Reenskaug : *Working with objects: A three-model architecture for the analysis of information systems.* JOOP May 1997.

©1998 Trygve Reenskaug numerica taskon Four Faces, *page 28*