

Modeling System Behavior
The What, the Why and the How of the UML Collaboration
Trygve Reenskaug
Version 003. Draft of May 29, 2000

<add something about reliability/simplicity>

Business information systems are becoming far too important to be left to the computer experts. Top management, marketing experts and the user community in general must be actively involved in shaping their future information systems. This means that they must not only know how to operate their current systems; they must also understand how the systems are structured so that they can distinguish between what will be easy to achieve and what will be hard. Only then can they be realistically creative.

It is never easy to make the complex appear simple without lying. So, enabling the users to become members of the exclusive body of information system cognoscenti presents a daunting challenge to the computer professionals. What is needed is a common language that lets the computer professional present the nature of an information system in terms that are meaningful and interesting to the user. And the presentation must be complete and true on the chosen level.

There is only one known language that can satisfy these needs. It is the language of objects. It is the UML Collaboration; a language that is standardized by the Object Management Group. The UML Collaboration lifts the CORBA descriptions of individual communication links between objects to descriptions of whole systems of interacting objects.

In the main body of this book, I present the language of objects as seen by the user community. Information systems are never quite what they appear to be. In an appendix, I show the computer expert how the illusion of objects translate into real computer systems.

Incidentally, computer experts will also find the language of objects ideally suited for expressing the architectures of distributed information systems. The language is therefore useful to professionals as well as laymen.

©2000 Trygve Reenskaug, Oslo, Norway.

All rights reserved.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made for profit or commercial advantage and that copies bear this notice and full citation on the first page

1 Introduction

The UML Collaboration is a powerful tool for modeling system behavior. It can be used for many purposes ranging from enterprise modeling and distributed system design to tracing requirements between different levels of abstractions and designing the details of an OO program. I will focus on the practical application of Collaborations and illustrate how they help us create simple solutions to complex problems through separation of concern. The book is based on UML 1.4 and includes both the collaboration and the collaboration instance constructs.

The intended audience for this book is senior programmers, information architects, students and managers who are interested in information systems architecture and development. Readers should be familiar with basic object oriented concepts.

The goal is that readers shall understand the syntax and the semantics of the UML collaboration language. They shall also understand where the collaboration is applicable and be able to teach it and apply it in practical modeling.

Reader's guide

Acknowledgements

etc.

2 What it is all about

All through the 1960's I was heavily engaged in developing a system for the computer aided design of ships. Through my shipbuilder friends, I became familiar with their systems for production planning and control. The kernel of their operation was a very sophisticated system for activity planning and resource allocation. The planning department was very happy with this system, and believed it to be very satisfactory for parties concerned. My line manager friends told a different story. What really happened was that the managers did their planning on the back of an envelope or with a private computer program. Once they had an acceptable plan, they diddled the input so that the planning department happily produced a plan they could accept.

The line managers had two main objections to the centralized planning system. First, this system had to rely on a common, albeit sophisticated, planning algorithm. So it could not take into account all the special considerations that applied to the different operations. The pipe shop, the panel assembly line and the big gantry crane spanning the dock area were instances of this. For example, a big panel had to be followed by at least one small panel on the panel assembly line. The gantry crane had two hooks for lifting. One hook could only lift light loads. The other hook could lift the heaviest loads up to 450 tons. So the crane could handle two independent loads as long as at least one of them was a light one. Clearly, no universal planning algorithm could handle issues like these.

Second, the centralized planning department was a staff function separate from the line organization. The planning system represented a compromise between many conflicting interests. The voice of an individual line manager was apt to drown in this situation. It would have been much better if authority over the planning operations could be distributed along with the authority over the ship construction operations in the ordinary line organization.

Our challenge was to devise a distributed system for planning and control that could be specialized according to all the different concerns and that could be owned by the line organization. By 1970, the ideas of object orientation as a means for mastering complexity reached my office across the yard from the Norwegian Computing Center by some mysterious process of diffusion. Objects seemed to answer the needs of shipyard planning because algorithms could be specialized and distributed together with the planning data.

The idea is illustrated in [figure 1](#). I envisioned the planning process as a negotiation between objects where each was responsible for a particular aspect of building a ship. For example, an *activity* involving lifting could ask the *gantry crane* for a reservation. The *crane* could then ask the *activity* about the weight involved and allocate a time slot accordingly.

Notice the use of words here. An activity cannot ask a crane for anything. But *objects representing them* can converse through message interaction. Further, there is nothing in the idea of objects that prevents these objects for being owned and controlled by different managers in the line organization.

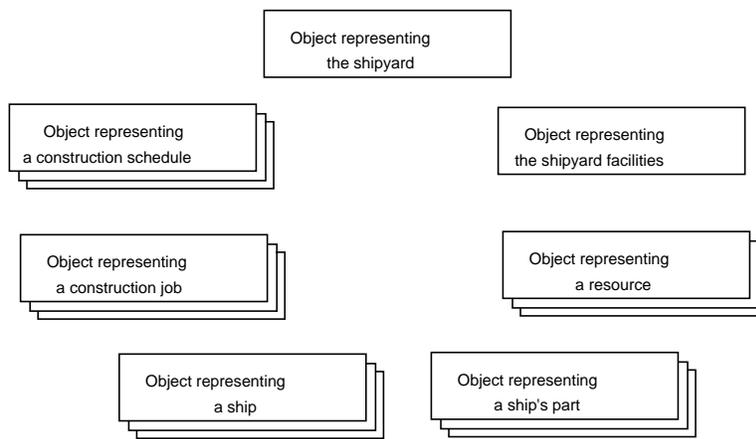


Figure 1. Some objects relevant to the shipyard scheduling and control operation

I clearly needed some way of describing a strategy for the negotiating objects that would lead them to a good overall plan. In 1977 I published a first attempt at describing how objects collaborate to reach a common goal [Reenskaug 77]. Figure 2 is taken from this paper and shows the overall system design.

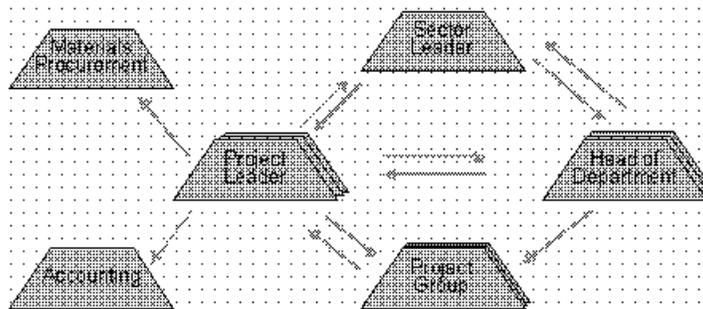


Figure 2. The world's first collaboration model; IFIP Conference, Toronto, 1977

The notation is archaic, but the semantics is still viable. In figure 3, the diagram is redrawn with the UML collaboration notation. An interesting feature of this collaboration is that these top level objects are directly tied to an owner in the line organization. The interaction diagram of figure 4 is part of a detailing of the ProjectLeader's component. The interaction was documented in a movie-style notation in the original paper. We here use the more familiar UML message sequence diagram and ignore the methods that were shown in the original notation. So we do not see that the Resource Requirement Component is doing the actual resource allocation.

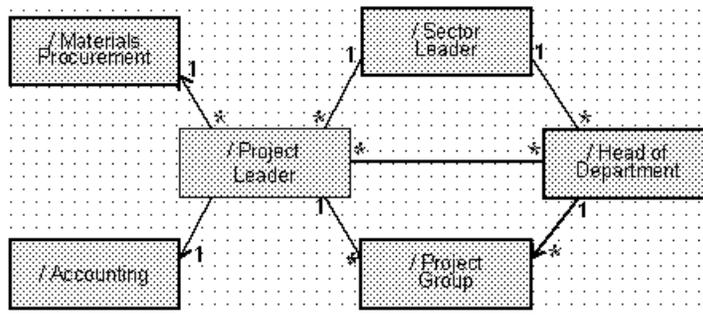


Figure 3. Example resource loading algorithm. Redrawn from the original.

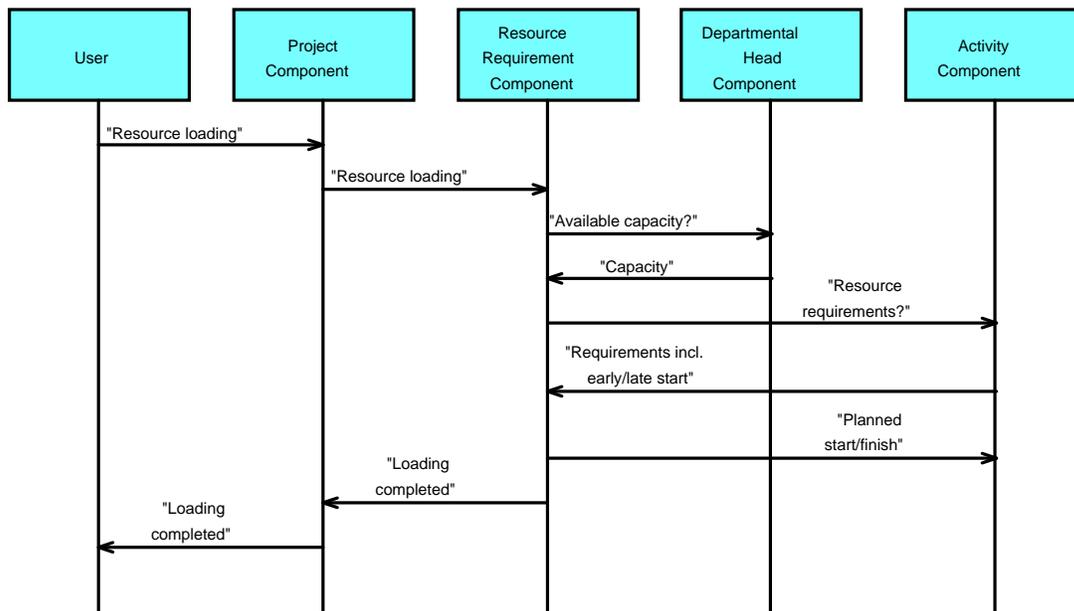


Figure 4. An early interaction diagram. Redrawn from the original.

An interesting aspect of this early collaboration diagram is that the objects are named by their owners in the shipyard line organization. The ruling idea was that the objects are robots that work on behalf of their owners and are completely controlled by them.

I first tried to use Simula for some early testing of my ideas. I was stumped. A serious problem was caused by the complexity introduced by the Simula typing system. Simula insisted that I knew the class of an object before I could send a message to it. But my objects should be able to negotiate with other objects regardless of their class as long as they behaved properly. Simula's insistence on common superclasses cluttered my programs and hindered my thinking.

So I went to the deplorable step of implementing a pre-processor to FORTRAN that made it somewhat object oriented. This helped me along for a bit, but it was clearly not a long term solution. By 1978 I had the good fortune to spend a year in the Smalltalk group at Xerox PARC . Smalltalk marked a major step ahead. I could now focus on the objects and relegate the classes to become an implementation detail.

As part of my current series of experiments, I have written a Java demonstration program that illustrates the ideas. If the technology cooperates, you should be able to run an Applet version in your web browser: [exampleActivityNetwork.html](#). Your screen should look somewhat like [figure 5](#). There are two built-in test networks: One with a common resource that can serve one activity at the time. Another that simulates the gantry crane where one activity can use a small hook while three others need the big one. Here is the source code if you should care to read it or even test it for yourself: [planningExample-0.zip](#).

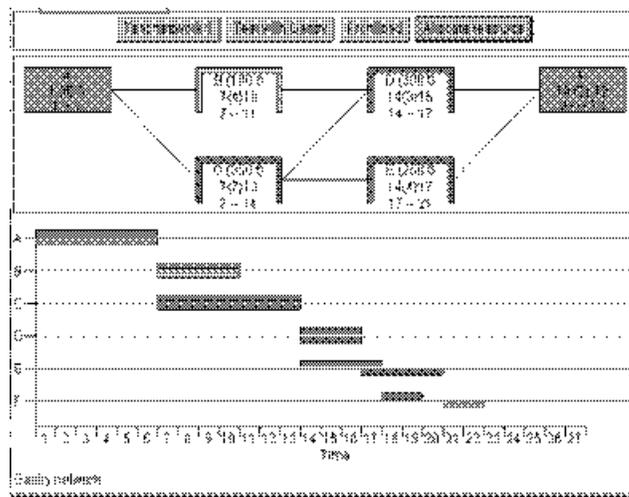


Figure 5. Example tool for activity network planning

The currently popular object oriented programming languages such as C++, Java and even Smalltalk have the common problem of limited scope. All objects exist within an object space that is strictly bounded by an execution of the run time system. Message interchange with objects in other object spaces is outside the scope of the programming language and is more in the nature of input/output.

To see a solution to these problems, I had to wait for the world of distributed objects as it was envisioned by the Object Management Group (OMG). OMG offers CORBA as a means for supporting interaction among huge structures of collaborating objects. The objects may reside on different hardware platforms with different operating systems, and the programs controlling the objects may be written in different languages. CORBA itself controls the links between the objects through the interfaces that define permissible messages on these links.

In my project for an object oriented activity planning system, I need to describe how a system of interacting objects cooperate to reach a common goal. The UML Collaboration offers a language for specifying and describing such systems. It builds on our OOram role modeling language which we developed through the eighties as part of our object oriented software engineering efforts.

You may well wonder what happened to our exciting product idea. It died somewhere along the line. The shipyard that was our sponsor and first customer suddenly lost their market due to some crisis in the oil industry. Their agenda changed overnight from optimizing their production of oil tankers to simple survival. The shipyard survived, but our project did not.

2.1 The dream of the bug free computer system

In a recent press campaign, the vendor of a leading operating system bragged that its new 2000-version had been extensively tested before release. This testing had been so thorough that no less than 60,000 bugs had been identified and corrected.

Edsger Dijkstra, one of the greatest brains in the history of computing, observed nearly 40 years ago that *testing can only show the presence of bugs*. Conversely, *testing can never show the absence of bugs*. [Dijkstra 1976]

A moment's thought makes one realize that the more bugs we find during testing, the more bugs there will be in the product we ultimately deliver to the unwary users. So if the excellent testing procedures of the above vendor finds 90% of the bugs, they left the remaining 6667 bugs to the entertainment of their customers.

Dijkstra's conclusion was that the only logical way to create software without bugs is to avoid introducing them in the first place! The key is to make the software so simple that it is obviously correct. What we need is *separation of concerns*. I quote from [Dijkstra 1976]:

"To my taste the main characteristic of intelligent thinking is that one is willing and able to study in depth an aspect of one's subject matter in isolation, for the sake of its own consistency, all the time knowing that one is occupying oneself with only one of the aspects. The other aspects have to wait their turn, because our heads are so small that we cannot deal with them simultaneously without getting confused. This is what I mean by 'focusing one's attention upon a certain aspect'; it does not mean completely ignoring the other ones, but temporarily forgetting them to the extent that they are irrelevant for the current topic.

Such separation, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts that I know of.

I usually refer to it as 'separation of concerns', because one tries to deal with the difficulties, the obligations, the desires, and the constraints one by one.

When this can be achieved successfully, we have more or less partitioned the reasoning that had to be done -- and this partitioning may find its reflection in the resulting partitioning of the program into 'modules' -- but I would like to point out that this partitioning of the reasoning to be done is only the result, and not the purpose.

The purpose of thinking is to reduce the detailed reasoning needed to a doable amount, and a separation of concerns is the way we hope to achieve this reduction.

The crucial choice is, of course, what aspects to study 'in isolation', how to disentangle the original amorphous knot of obligations, constraints and goals into a set of 'concerns' that admit a reasonably effective separation.

To arrive at a successful separation of concerns for a new, difficult problem area will nearly always take a long time of hard work; it seems unrealistic to expect otherwise.

The knowledge of the goal of 'separation of concerns' is a useful one: we are at least beginning to understand what we are aiming at." [Dijkstra 1976]

Dijkstra puts it all in a nutshell: "

"There are two ways of constructing a software design:

- One way is to make it so simple that there are obviously no deficiencies*
- and the other way is to make it so complicated that there are no obvious deficiencies."*

Dijkstra used the principle of separation of concerns to simplify complex systems. With structured programming, an initially complex algorithm is decomposed into a sequence of simpler steps, each with a well-defined purpose. A similar technique is called *functional decomposition*. Hardened

programmers are forever grateful to Dijkstra for liberating them from the spaghetti code caused by the indiscriminate use of *goto* statements

While structured programming is excellent for simplifying complex algorithms, it does not help simplify complex information structures. The technique of *semantic modeling* confronts the problem of complex data with simple logic. The assumption is that once we get the data structures right, any number of almost independent programs can work against the common database containing the shared data.

Modern distributed systems frequently combine complex logic with complex information structures. An answer to this challenge is to lump chunks of information with the associated logic into *objects*. The complex information structure is mapped onto a *class structure*. The concept of class inheritance invites a common description of common code while specializations of the classes takes care of special requirements that apply to different variants of the objects.

The centralized logic of traditional programs is distributed to the different classes so that each object becomes responsible for its natural part of the overall logic. Readers knowing some object oriented programming language such as Java will recognize that the objects are specified in corresponding Java *classes*; the information stored in each object is specified as class *attributes*; and the various chunks of logic is specified in corresponding *methods*.

The previous functional decomposition is apparently lost, because each object realizes some small part of every function. [Figure 6](#) illustrates the two-dimensional nature of the problem.

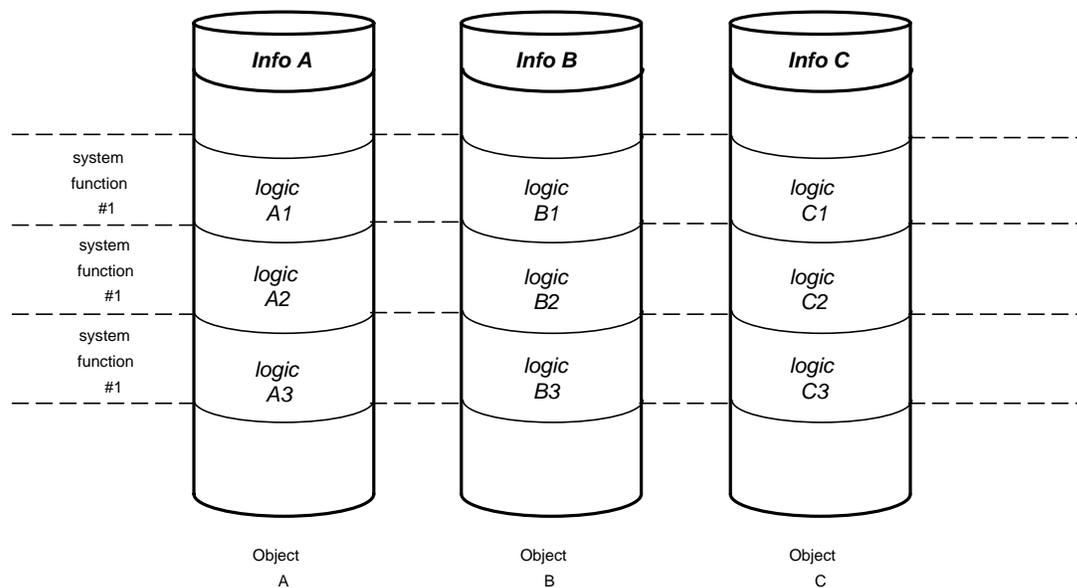


Figure 6. The two dimensions of complex information structure combined with complex logic.

Currently popular programming language have no way of partitioning the logic in accordance with the system functions. Such partitioning is precisely the purpose of collaboration modeling where we answer the following questions:

1. What is the overall purpose of the system ?
e.g., *system function #1*
2. What are the objects ?
e.g., *Objects A, B, C*

3. What are their responsibilities ?
e.g., *take care of logic B1 (part of system function #1)*
4. How are they interconnected ?
Figure 3 shows an example.
5. How do they interact ?
Figure 4 shows an example.

We will consider three techniques for the separation of concerns in the following. The most important one is based on systems thinking and separation on system function. (Also called Use Cases). The second is the *component*, which is an object that offers a general set of services to its possibly distributed environment. The third is the *framework*, which is a collaboration that realizes a basic system function in a way that is intended to be specialized for different purposes.

2.2 The dream of putting the user in the driver's seat

In his book *The design of everyday things*, Donald A. Norman laments the paucity of the design of the multitude of different things that surround us in our daily life. One of his main tenets is that a thing should be immediately usable without a user's manual. A recurring theme throughout the book are two fundamental principles of designing for people: (1) provide a good conceptual model and (2) make things visible. "*A good conceptual model allows us to predict the effects of our actions. Without a good model we operate by rote, blindly; we do operations as we are told to do them; we can't fully appreciate why, what effects to expect, or what to do if things go wrong*".
[Norman]

Norman discusses everyday things like doors and light switches and telephones and automobiles. But I can hear your violent objection: "*Surely, a computer application cannot be so obvious that it can be understood without a manual?*" Personally, I would add that I know systems where even the manuals are completely incomprehensible.

The key is that the thing or application or whatever has to be *designed* for comprehensibility. For success, the designer has to build on the knowledge of the prospective user and on knowledge that already exist in the culture and the environment. See Norman's book for a deeper discussion of these issues.

It is always hard to communicate new concepts and ideas to an unprepared audience. I have more than once been frustrated when trying to explain new technologies and opportunities to an audience of prospective users. In spite of all my efforts for precision and sobriety, half the audience become firmly convinced that all problems will now be solved. The other half takes the opposite view and writes the whole proposition off as so much hot air.

Figure 7 illustrates the distortion that is inherent in all human communication. *Person A* wants to communicate *Meaning-A* to *Person-B*. He codifies this meaning according to his personal language and vocabulary; *Language-A*. The result is *data* in the form of sound waves or whatever. *Person-B* senses these data and decodes them according to his or her personal vocabulary, *Language-B*, to get the underlying information, *Meaning-B*. Since *Language-A* is more or less different from *Language-B*, *Meaning-B* will be more or less different from *Meaning-A*. This explains why different parts of my audience interpret what I say in different ways. It also explains why only a few of them get my meaning exactly as I intend it.

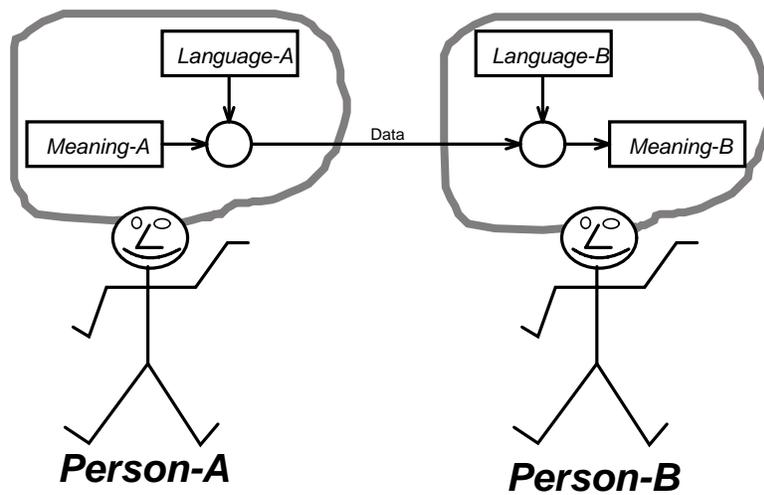


Figure 7. The human communication process

The key to effective communication is, therefore, that we have a common language so that we can formulate our ideas and share them. I suggest that *the language of objects* is a good candidate for universal communication about computer based systems. I will also endeavor to show that it is the ideal language for discoursing about *distributed* and *component based* systems.

Let us first look at some examples where people have attempted to make unknown things clear to their prospective users.

Whenever we get a new technology, the populace needs to internalize an understanding of the new and mysterious things. Some simply reject it and cling to the old ways. Some of us may remember old people who refused to have anything to do with the telephone. Some more adventurous tried it, but were firmly convinced that they had to shout at the top of their voice to be heard over the long distance telephone wires.

For every new technology, benevolent people try to explain the new technology so that it can be understood by the meanest intelligence. I have a delightful book from 1911 called *The Electricity*. It lays out the wonders of this mysterious new fluidum. At an electric exhibition in London in 1905 they served a dinner for 65 persons. All the courses were boiled or fried by electricity; and it was all done within the dining room itself as shown in [figure 8](#). In the words of *The Electricity*: "*In our modern times, electric cooking is by no means limited to such exhibitions. Both in hotels and private homes around the world there exist electric kitchens where the heat is extracted from electric wires.*"

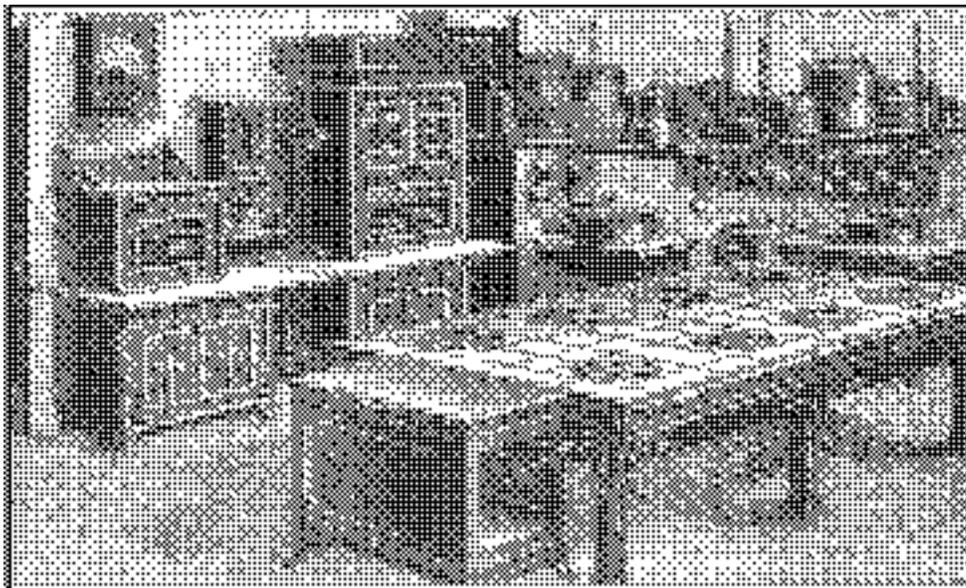


Figure 8. Electric cooking at the 1905 electric exhibition in London.

The authors give a number of examples of the wonders of electricity. As is so often the case, the author sometimes gets carried away by his enthusiasm. [Figure 9](#) shows an electric taxi. It was described as having enormous advantages over the noisy and stinking benzine driven alternative. It was admitted that the cost and weight of the batteries posed a problem. In 1911, it was confidently expected that these problems would all be solved in the near future. Have you heard this one before?

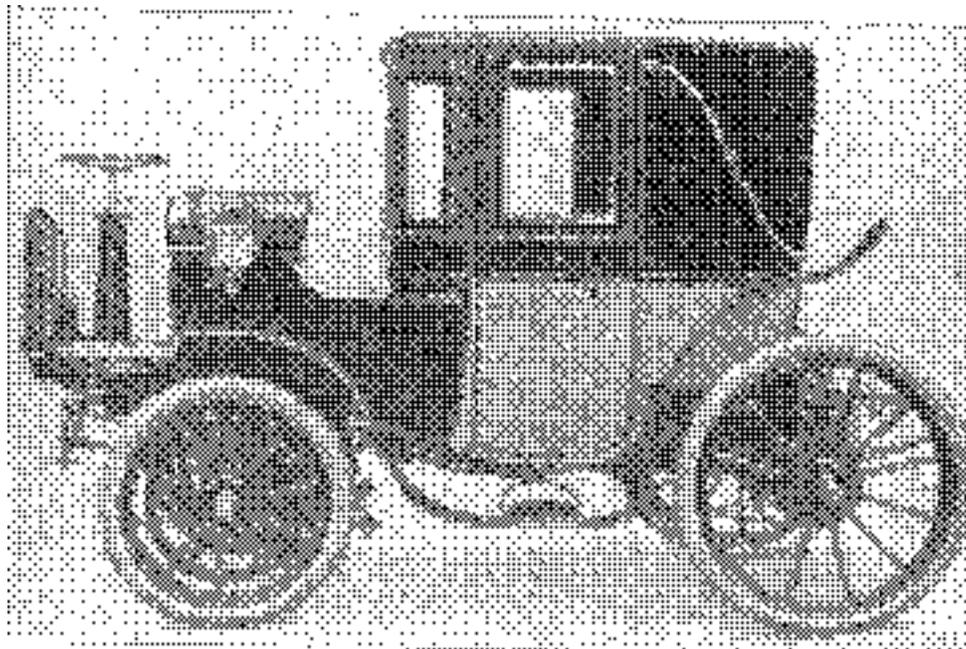


Figure 9. An electric taxi.

This story reminds me of a guest lecture at the University of Oslo in 1962. It was given by one of the great gurus of artificial intelligence who shall remain unnamed. He showed us how he and his team had developed a robot that could see and analyze a scene so that it could build a tower from

children's bricks. He confidently predicted that within five years, his group would have a robot with the intelligence of a five year old child. Have you heard this one before?

I assume most readers know the Lego® bricks that children use to construct a wide variety of interesting things. Electric motors, wheels, toothed wheels, belts and pulleys were added to the range of pieces several years ago. A recent addition is a robot controller together with sensors and actuators. The controller is actually a computer that can be programmed by the children through a programming environment on a PC. The controller brick is shown in [figure 7](#).

At the top is a gray row of three contacts for sensors named 1, 2, and 3. The sensors are specialized bricks that are sensitive to touch, light, etc. The sensors have connecting wires that end in a plug that is interchangeable with normal Lego bricks. Any sensor can be plugged into any position.

Towards the bottom is black row of three contacts named A, B, and C for actuators such as motors, lights, etc. Any actuator can be plugged into any of these positions.

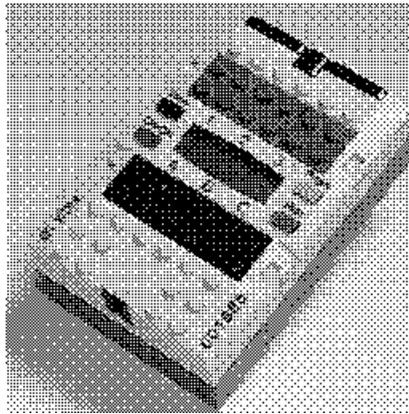


Figure 10. The Lego Mindstorms RCX; a Programmable Brick.

I learned about this brick from Oddbjørn, my 9 year old grandson. He had programmed a robot that would wave its arms and stop if the arm hit something solid. He also showed me the programming environment and demonstrated how he transferred programs to the controller through an infrared link.

The programming environment is shown in [figure 8](#). Programming is done by composing modules called *bits*. In the illustration below, we see that a light sensor is attached to plug #2. 'Dark' is set as a brightness from 0 through 45, while 'bright' is set as a brightness from 46 through 100. When the sensor is dark, actuator A shall be switched on and actuator C shall be switched off. Conversely, if the sensor sees a bright light, actuator C shall be off and A shall be on. So using this program, Oddbjørn can make his robot do different things when he flashes a light at it.

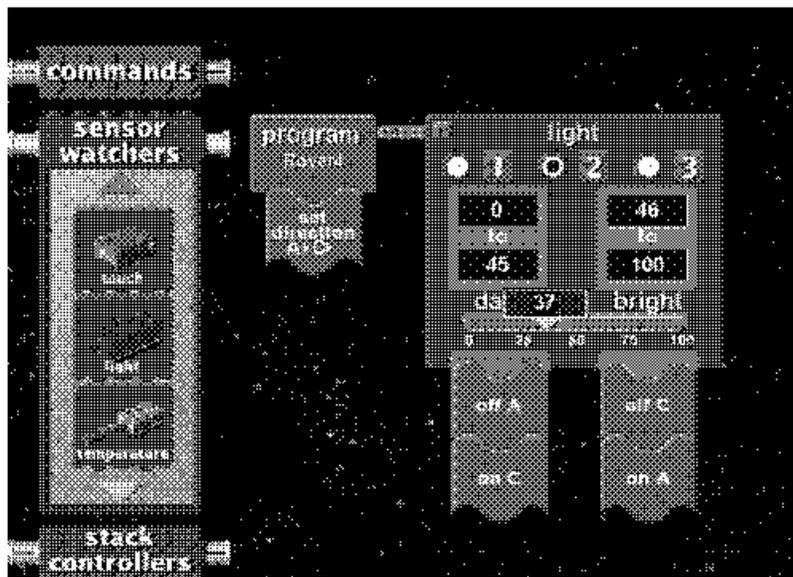


Figure 11. Lego Mindstorms

Children are used to doing things that they do not quite understand. So Oddbjørn works happily with this programming environment and keeps returning to the program to figure out why his robot doesn't behave as he intended.

So what about grown-ups? Several elements have to be in place for computer applications to become doable and understandable, preferably without manuals.

1. First, there must be a common language that exists in the culture and is reasonably well understood by everybody. I propose that the language of collaborations is a candidate for such a language.
2. Second, designers must provide a model of their product that is expressed in this language.
3. Third, implementors must maintain the illusion that this model is indeed a true model of the product.
4. Fourth, the model must be made visible and tangible through the user interface so that users can relate to it and manipulate their information within the context of the model.
5. The language of collaboration holds the promise of being extended to a user's programming language some time in the future. It could be something similar to Lego Mindstorms, but vastly more powerful.

As an example, consider the spelling checker in MS Word®. I have read somewhere that in this product, language checking is done in a separate component as illustrated in [figure 12](#). Assume that we were really living in a true universe of objects. The user community could easily understand the nature of the system. So, when they were to specify a new order entry system, they could immediately see that it would be easy to add spell checking to a text input field in the new order entry system. Psychologists tell us that we only tend to wish for things that we know are feasible. So an understanding of the construction of the current systems enables the user community to be creative in the specification of new systems.

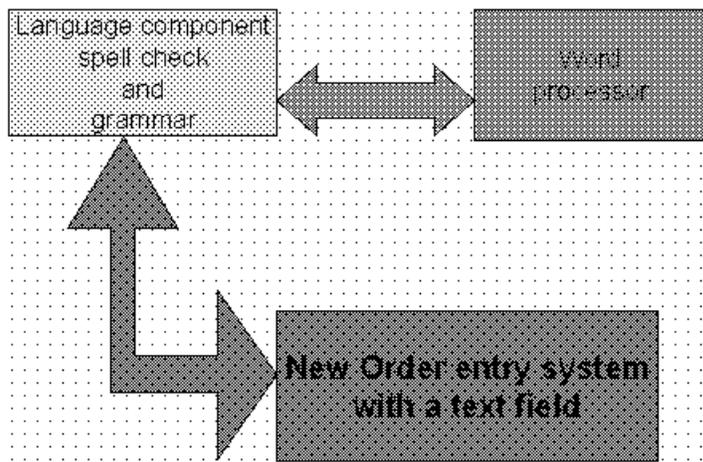


Figure 12. An object based language component can easily be reused

It is interesting to note that the needs of the systems architect, designer, and implementor for simplicity and separations of concern very nicely coincide with the needs of the user community for a modeling language that leads to simple and understandable models.

2.3 The OMG dream of a world of objects

The Object Management Group (OMG) was founded in April 1989 by eleven companies, including 3Com Corporation, American Airlines, Canon, Inc., Data General, Hewlett-Packard, Philips Telecommunications N.V., Sun Microsystems and Unisys Corporation.

The OMG was formed to create a component-based software marketplace by hastening the introduction of standardized object software. The organization's charter includes the establishment of industry guidelines and detailed object management specifications to provide a common framework for application development. Conformance to these specifications will make it possible to develop a heterogeneous computing environment across all major hardware platforms and operating systems. Implementations of OMG specifications can be found on many operating systems across the world today. OMG's series of specifications detail the necessary standard interfaces for Distributed Object Computing. Its widely popular Internet protocol IIOP (Internet Inter-ORB Protocol) is being used as the infrastructure for technology companies like Netscape, Oracle, Sun, IBM and hundreds of others. These specifications are used worldwide to develop and deploy distributed applications for vertical markets, including Manufacturing, Finance, Telecoms, Electronic Commerce, Realtime systems and Health Care.

The OMG vision is set out in their Architecture Guide [OMG 95]. Extracts:

"The CPU as an island, contained and valuable in itself, is dying in the nineties. The next paradigm of computing is distributed or cooperative computing. This is driven by the very real demands of cooperation recognizing information as an asset, perhaps their most important asset."

"To make use of information effectively, it must be accurate and accessible across the department, even across the world. This means that the CPUs must be intimately linked to the networks of the world and be capable of freely passing and receiving information, not hidden behind glass and cooling ducts or the complexities of the software that drives them."

The members of the Object Management Group have the shared goal of developing and using integrated software systems. These systems should be built using a methodology that supports modular production of software; encourages reuse of code; allows useful integration across lines of developers, operating systems and hardware; and enhances long-range maintenance of that code. Members of the OMG believe that the object-oriented approach to software construction best supports their goals."

The idea of an object forms the foundation for the OMG universe of collaborating objects:

"An object can model any kind of entity or concept; for example, a person, a ship, a document, a department, an order transaction, ..." "A basic characteristic of an object is its distinct object identity, which is immutable, persists for as long as the object exists, and is independent of the object's properties or behavior."

"Operations are provided by an object."..."Each operation has a signature. A signature consists of the operation's name, set of parameters, and set of results."..."In its simplest form an operation need not have parameters and need not return a value."

2.3.1 The object structure as a rational organization

Objects may alternatively be defined in terms of a programming language such as Java. I prefer the more direct metaphor mapping the object system to a formal business organization. This metaphor has the added advantage of coinciding with OMG idea of objects. I think of objects as mechanical clerks as illustrated in [Figure 13](#). Each clerk has an *in-basket*, a *private data file*, a *private book of rules*, and one *out-basket* for each of its collaborators.

Clerks cooperate through message interaction. A message is like a business form. It consists of the name of the message type together with actual values. A clerk picks up a message from his in-basket and processes the message according to an appropriate rule selected from his book of methods. Processing may include reading and modifying values in the private data file as well as sending messages to one or more collaborator clerks.

Objects are ideal building blocks for representing complex systems because they combine information with behavior. I regard the essence of object orientation to be the modeling of interesting phenomena as structures of interacting objects, and I always consider objects in the context of their collaborators.

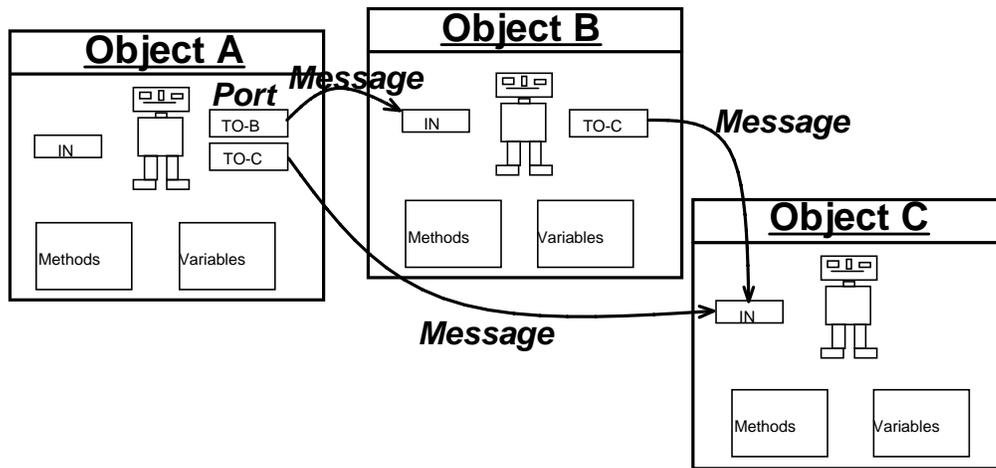


Figure 13. A fruitful object metaphor

Our simple metaphor highlights important aspect of objects and object modeling:

1. Objects are *encapsulated*. Objects (clerks) communicate through message interaction only. The in-basket, the out-basket, the file, and the book of rules are private to the object and invisible to other objects. This enables us to design the object organization as a whole without worrying about the details of the object implementation.
2. Objects exhibit *polymorphism*. Since each object (clerk) has its own book of methods, different objects may handle messages differently according to their individual characteristics. This gives us the freedom to be goal oriented: We focus on what we want to achieve rather than how it is done. Each object can be designed to do the right thing, the exact details depending on the nature of the object.
3. Objects, like people, have *identity*. An object (clerk) retains its identity throughout its lifetime. There has never been and will never be another object with the same identity. This is essential for reasoning about object collaboration: who does what, why do they do it and when do they do it?

Anybody can participate in the design of an object oriented system because of this analogy to the design of a business organization. The mechanical clerks do exactly as they are told; there is no common sense or uncalled for initiative.

The clerk analogy is very useful for designing object-oriented systems: given a task, what is the optimal organization of trusted clerks that will perform it? We endeavor to organize so as to get a clear division of authority and responsibility; to minimize duplication; to simplify communication; and to employ similar objects in different positions in order to reduce the need for programming.

The importance of object identity in computers and organizations is illustrated by this old and delightful story:

This is a story about four people named Everybody, Somebody, Anybody and Nobody. There was an important job to be done and Everybody was sure that Somebody would do it. Anybody could have done it, but Nobody did it. Somebody got angry about that, because it was Everybody's job. Everybody thought Anybody could do it, but Nobody realized that Everybody wouldn't do it. It ended up that Everybody blamed Somebody when Nobody did what Anybody could have done.

More than 100 years ago, Max Weber presented his vision of the rational work organization. The following literal excerpts from [Etzioni 64] form a beautiful description of this "perfect bureaucracy": logical, rational, extremely efficient, and extremely rigid. Applied to the automated object system, it is perfect. Here Weber's rules with my comments in parenthesis:

1. *Emphasis of structure.* "A continuous organization of official functions bound by rules." *Rational organization is the antithesis of ad hoc, temporary, unstable relations; hence the stress on continuity. Rules save effort by obviating the need for deriving a new solution for every problem and case; they facilitate standardization and equality in the treatment of many cases.* (Focus on the objects and their formal responsibilities. Ignore all informal contacts and interactions.)
2. *A specific sphere of competence.* "This involves (a) a sphere of obligations to perform functions which have been marked off as part of a systematic division of labor; (b) the provision of the incumbent with the necessary authority to carry out these functions; and (c) that the necessary means of compulsion are clearly defined and their use is subject to definite conditions." *Thus a systematic division of labor, rights and power is essential for rational organization. Not only must each participant know his job and have the means to carry it out, which includes first of all the ability to command others, but he also must know the limits of his job, rights, and power so as not to overstep the boundaries between his role and those of others and thus undermine the whole structure.* (Focus on the formal object responsibilities, attributes and actions.)
3. *Hierarchy.* "The organization of offices follows the principle of hierarchy; that is, each lower office is under the control and supervision of a higher one." *In this way, no office is left uncontrolled. Compliance cannot be left to chance; it has to be systematically checked and reinforced.* (Well, we are not quite so rigid, but we carefully control the collaborators of an object and the messages it may send to them.)
4. *Norms of conduct.* "The rules which regulate the conduct of an office may be technical rules or norms. In both cases, if their application is to be fully rational, specialized training is necessary. It is thus normally true that only a person who has demonstrated an adequate technical training is qualified to be a member of the administrative staff..." (This is strictly true, the system is completely defined by the program. Perhaps we should add that the system should be documented with a UML model.)
5. *Independence.* *In order to enhance the organizational freedom, the resources of the organization have to be free of any outside control and the positions cannot be monopolized by any incumbent. They have to be free to be allocated and re-allocated according to the needs of the organization. "A complete absence of appropriation of his official positions by the incumbent" is required.* (This is the principle of object encapsulation. An object should only influence other objects through their official interfaces. No hidden or "smart" couplings between objects!)
6. *Documentation.* "Administrative acts, decisions, and rules are formulated and recorded in writing..." *Most observers might view this requirement as less essential or basic to rational organization than the preceding ones, and many will point to the irrationality of keeping excessive records, files, and the like, often referred to as 'red tape'. Weber, however, stressed the need to maintain a systematic interpretation of norms and enforcement of rules, which cannot be maintained through oral communication.* (A trivial fulfillment of this rule is through the program source code. We read the rule to mean that we also want higher level documentation describing goals, specifications, architecture and design.)

2.3.2 Modeling systems of collaborating objects

Max Weber's rational organization is a metaphor for the distributed information system. [Figure 14](#)

illustrates the OMG vision of a universe of collaborating objects. Each circle denotes an object. The real communication structure will be much more complex than the structure indicated in the figure.

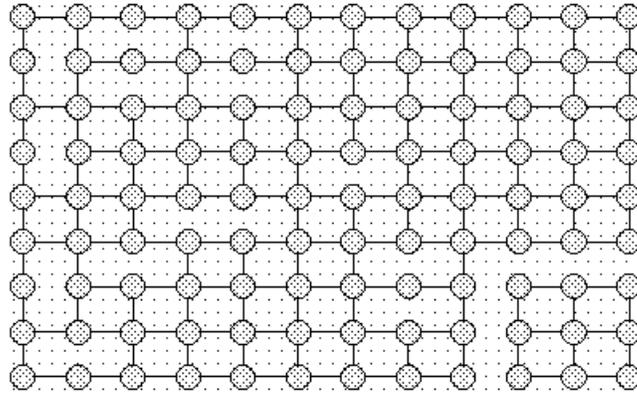


Figure 14. The OMG vision of a Universe of Objects

Consider our vision for a shipyard information system as illustrated in [figure 3](#). Each circle in [figure 14](#) would represent one of the shipyard's information objects. Some would represent construction projects; others would represent activities, resources, materials, etc. The total universe of objects is clearly far too complex to be grasped by our mind.

OMG's initial contribution to our mastering of this complexity is to focus on the control of *individual interfaces* as indicated in [figure 15](#).

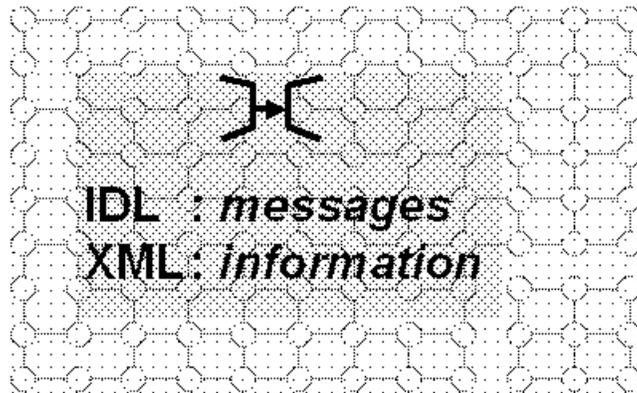


Figure 15. CORBA controls the Interfaces

OMG supports two standards that are relevant to interface control:

1. The *Interface Definition Language*, [[IDL](#)] is used to specify interfaces that describe the signature of all messages that an object is permitted to send to another object along a given link. For example, the Java message
`public void frontLoad (int time)`
looks like this in IDL
<to be added later>

2. The *Extensible Markup Language [XML]* is a language for encoding information. A number of groups are hard at work defining the encoding of information for a great variety of domains. Here is a possible codification of an Activity object:

```
<activity>
  <name>Activity-B</name>
  <duration>4</duration>
  <earlyStart>7</earlyStart>
  <earlyFinish>10</earlyFinish>
</activity>
```

The CORBA focus on the single link interfaces is very narrow and we cannot describe how objects cooperate to reach a common goal. The interface is too narrow, the whole is unmanageably large. What we need is a capability for studying a part of the whole; large or small as the need arises. Holbæk-Hansen's notion of a *system* helps us focus on some aspect of the whole:

"A *system* is a part of the world which we choose to regard as a whole, separated from the rest of the world during some period of consideration, a whole which we choose to consider as containing a collection of components, each characterized by a selected set of associated data items and patterns, and by actions which may involve itself and other components." [Hol 77].

<Should be rephrased to conform to current terminology>

Figure 16 illustrates how we choose some interlinked objects to constitute a *system* that we want to study for some purpose. The figure shows three, partly overlapping systems marked red, green and blue.

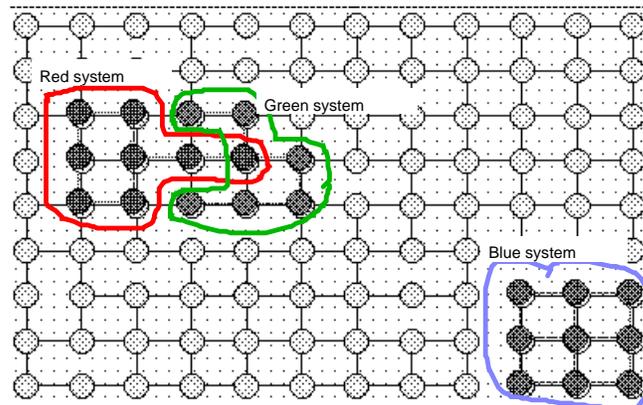


Figure 16. A system is a part of the real world which we choose to regard as a whole.

We have solved one problem and added another. How do we choose wisely? What should be included in a system and what should be excluded? Here are some alternative criteria that could be used alone or in combination:

1. As programmers, we may be tempted to choose all instances of a given class or the instances of a few classes. For example, we could choose all Activity objects and/or all Resource objects. This could be less than informative because activities from all projects would be mixed together with all resources. There could even be objects for resources that were not used by any activity at all.

2. We could separate on *function*, also known as *use cases*. This was the idea illustrated in [figure 5](#). We could start with the object for a given project and then describe systems for various interesting functions such as:
 - *UC 1 Frontloading*: Compute the earliest start time for all activities given the start time for the start activities. Assume unlimited resources.
 - *UC 2 Backloading*: Compute the latest end time for all activities given the end time for the end activities. Assume unlimited resources.
 - *UC 3 Resource allocation*: Allocate resources to all activities and thus produce a realistic schedule for the project within the given resource limitations.
3. We could separate on *ownership*. Model a system consisting of all objects that belong to a given part of the line organization. For example, model all objects that are relevant to the pipe shop. This criterium could be very interesting in these days of outsourcing. Next year, the pipe shop could be fissioned out as a separate company. It would then be very likely that they would then insist on controlling their own information systems, thank you.

Whatever our choice, we have solved one problem and created a new one. We can choose our system to be sufficiently restricted in scope so that we can understand it. But the systems will not be independent, so how do we manage their interdependencies?

The concept of *open systems* answers this problem:

For a given system, the environment is the set of all objects outside the system who affect the system or who are affected by the system. [Etzioni]

The blue system of [figure 16](#) is not linked to the universe of objects around it. It is therefore a *closed system*.

The other two systems are linked to their environment. The red system, for example, has 10 objects in its environment as shown in figure 18. One of them is part of the green system. This common object forms a link between these two systems.

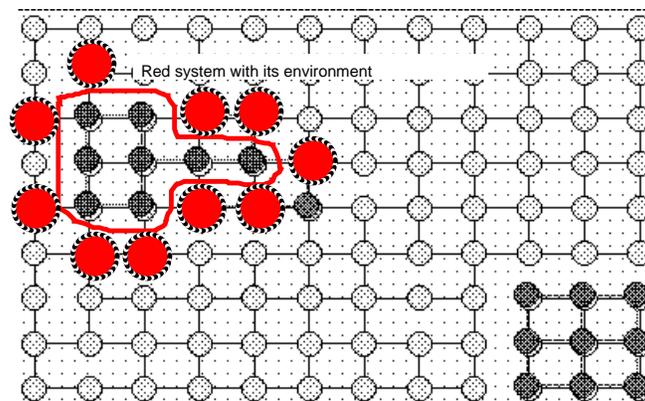


Figure 17. The Red system has objects in its environment.

Let's consider the three use cases mentioned above. Figure 13 shows the corresponding UML Use Case Diagram. The system is shown as a rectangle. Each Use Case is identified by an ellipse within the system rectangle. The environment objects are known as *Actors* and are shown outside the system.

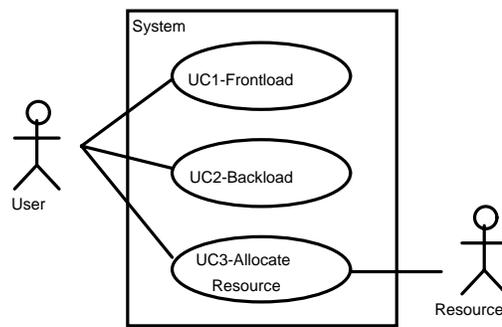


Figure 18. Use Case diagram for shipyard scheduling

Quite neat. We have tidied the OMG universe of objects to the UML idea of Use Cases through the concept of open systems.

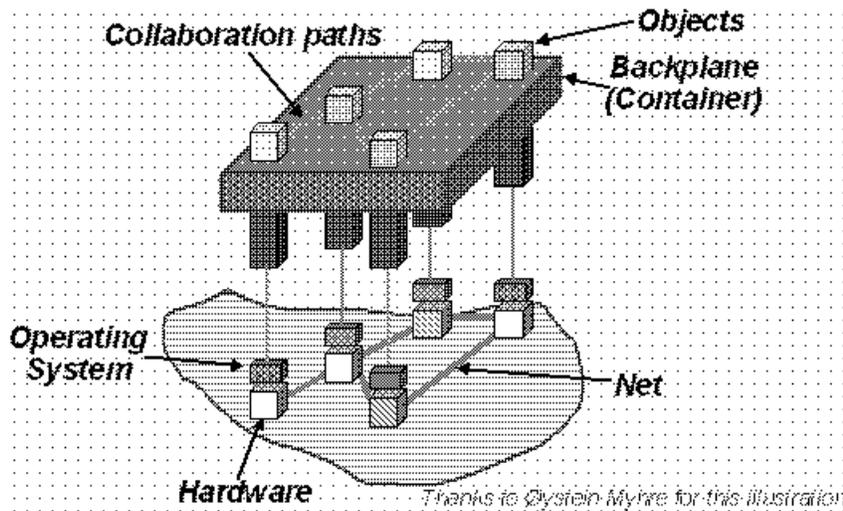


Figure 19. Object pluggable on a backplane. Platform with run time system, communication software and hardware hidden below the backplane.

Focus on:

1. Objects !
2. Responsibilities !
3. Message paths !
4. Interfaces !
5. Collaborations !

Make invisible:

1. Code
2. Communication
3. etc.

Challenge: Manage complexity

Solution: Objects & Collaborations

As an added benefit, we have liberated ourselves from the tyranny of classes. Remember the requirement for heterogeneous systems. The objects belonging to the outsourced piping shop could be coded in a different language from all the other objects. And we can still reason about the interaction between all the objects!

Enjoy.

3 Modeling with objects

UML is a *modeling language* and should not be confused with a programming language. Granted that many modeling tools offer an automatic code generator. But these tools can only generate skeleton code. Many details cannot be generated simply because the necessary information is not represented in the model. This is as it should be because any abstraction is based on a choice of what is considered important and what can safely be ignored. So a model a model is never complete, but focuses on essentials and hides inessentials. The choice of what is essential is critical because a good choice will help us think useful thoughts.

The highly successful and useful class abstraction is the result of one such choice. It is useful for modeling many interesting aspects of objects. UML 1.3 defines the notion of *class* as follows:

class [UML 1.3 glossary]

A description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment.

An example illustrates what this definition covers and what it doesn't cover. The class diagram of [figure 20](#) illustrates how a *Person* is the child of a *Man* and a *Woman*: A *Person* has exactly one father and one mother. A *Man* can have many children, and so can a *Woman*. Further, *Man* is a special kind of *Person* just as *Woman* is a special, but different, kind of *Person*.

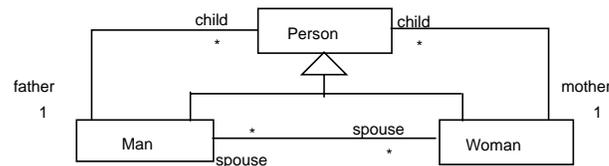


Figure 20. A class diagram

A very powerful model capturing a world of information with three simple classes. The number of instances of class *Person* in the world today exceeds 6 billion. In addition, there are all the people of the past. Should we guess 9 billion instances all told? About half of them are also instances of class *Man*, another half instances of class *Woman*. A person has exactly one *mother* and one *father*; both men and women can have many children. The many-to-many association between *Man* and *Woman* indicates that both a *Man* and a *Woman* can have any number of spouses; not necessarily at the same time.

The Entity Relation model is a well known abstraction. It is an excellent example of a successful separation of concerns because it highlights information structure and hides all considerations of behavior. The UML class abstraction is reminiscent of the Entity Relationship abstraction. The *class* corresponds to the Entity while the *association* corresponds to the Relation. In addition, the class models behavior in the form of operations that can be performed by its instances.

The simple family model of [figure 20](#) can be implemented in many different ways. Here are two possibilities. Both illustrate that the class model considers the static information structure as being important:

Alternative 1: Implement with instance variables

```
public class Person {
    public Person[] child;
    public Woman mother;
    public Man father;
}
public class Man extends Person {
    public Woman[] spouse;
}
public class Woman extends Person {
    public Man[] spouse;
}}
```

Alternative 2: Implement with reference methods

```
public class Person {
    public Person[] child();
    public Woman mother();
    public Man father();
}
public class Man extends Person {
    public Woman[] spouse();
}
public class Woman extends Person {
    public Man[] spouse();
}}
```

Now let's extend the model with some behavior. Consider that a person can ask for and receive money. We add the operations *cashRequested()* and *cashReceived()* as shown in [figure 21](#). We further assume that all persons are interested in money and add an attribute *cashInHand*. Since this is an attribute of class *Person*, all men and women will have inherited this attribute from class *Person*.

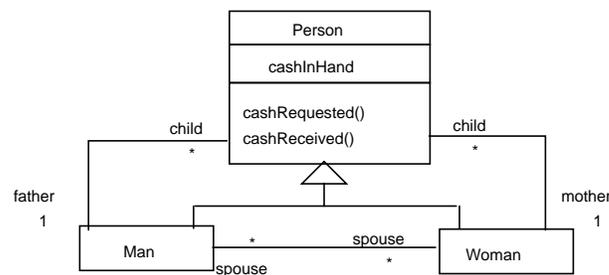


Figure 21. A class diagram with attributes and operations

The model specifies two additional methods in class *Person*:

Alternative 1: Implement with instance variables

```
public class Person {
    public Person[] child;
    public Woman mother;
    public Man father;
    public Currency cashRequested();
    public Currency cashReceived();
}
public class Man extends Person {
    public Woman[] spouse;
}
```

```

public class Woman extends Person {
    public Man[] spouse;
}

```

Alternative 1: Implement with instance variables

```

public class Person {
    public Person[] child();
    public Woman mother();
    public Man father();
    public Currency cashRequested()
    public Currency cashReceived()
}
public class Man extends Person {
    public Woman[] spouse();
}
public class Woman extends Person {
    public Man[] spouse();
}

```

The class abstraction deals in sets of Instances and sets of Links. This makes it hard to identify the sender and the receiver of a message. In the model above, we describe that a person can own cash; receive cash, and be asked for cash. But we cannot describe that a specific person gives cash to another person because there is no way of identifying a particular giver-receiver pair among the many billion of possible pairs.

Despite its undisputed success, the class abstraction is based on a *choice* of what is essential and what isn't. It is both possible and useful to make other choices that will lead to other abstractions. In particular, the Collaboration abstraction models object identity and message sending. This makes it useful for modeling the behavior of a system of collaborating objects since we can model both the senders and the receivers of messages.

Figure 22 illustrates how the collaboration models a system of interacting objects.

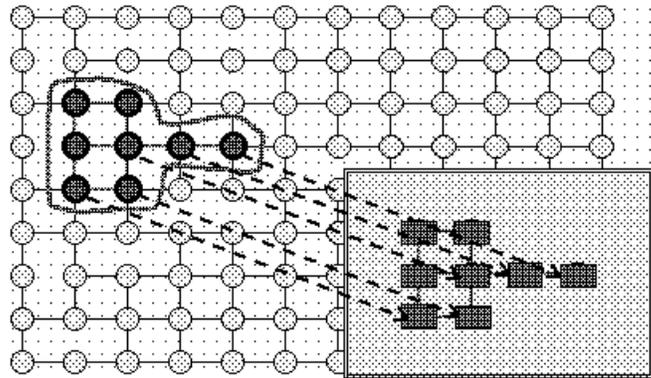


Figure 22. The UML collaboration models a system of objects by the roles they play in the system

Figure 13 is an "artist's impression" of an instance level collaboration. Figure 23 shows the same model in UML notation.

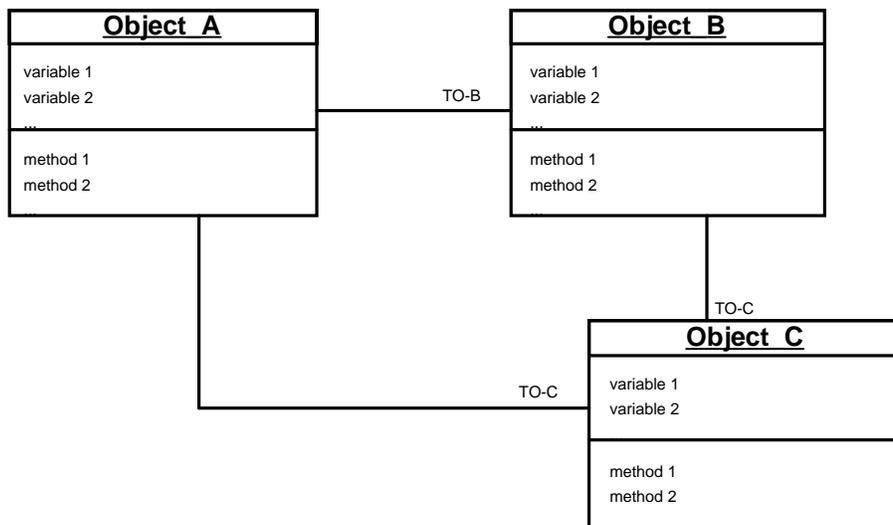


Figure 23. *Figure 13 redrawn according to the UML notation.*

UML uses rectangles for many purposes, objects being one of them. The name is underlined to show that objects are meant here. The lines connecting the objects denote message links. Somewhat perversely, the name of an object's reference to another object is shown on the far end of the line as shown in figure 23.

Here's a simple example of how we can describe system behavior with a UML Collaboration. Consider a Collaboration with three objects: My father, my mother, and myself. The names of these objects happens to be Bjarne, Gina, and Trygve as shown in figure 24:

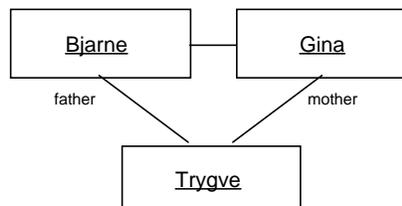


Figure 24. *A collaboration*

Notice the analogy with our simple object metaphor in figure 13. A rectangle represents an object, i.e., a clerk. The lines represent links for transferring messages. The line ends marked 'father' and 'mother' represents Trygve's out-baskets.

It is now meaningful to model that I ask my mother for some cash and get it. The *interaction diagram* in figure 25 shows a possible system behavior:

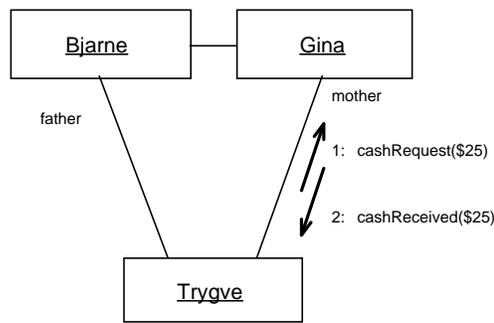


Figure 25. A simple interaction

This is beautifully concrete. There is no doubt as to who I ask for money, and the money is paid to me, not my brother.

The whole is greater than the sum of its parts. So the focus is on the collaboration as a whole because it has a value that exceeds the collective value of the objects taken separately. The important questions that can be asked from a collaboration are questions such as "What does it achieve?", "What are its objects?", "What are their responsibilities?", and "How do they interact?". A question that is of secondary importance is "How are its objects constructed?". This question is answered by giving a reference to the appropriate class or classes. Conversely, a class description can include a reference to relevant ClassifierRoles and their Collaborations.

Compare the definition of *collaboration* with the definition of *class* as given above:

Collaboration

A Collaboration describes how a number of objects work together for a common purpose. There are two aspects. The structural aspect is a description of the responsibilities of each object in the context of the overall purpose of the collaboration; and also the links that connect the objects into a communicating whole. The dynamic aspect is a description of how stimuli flow between the objects to achieve the common purpose.

It was easy to derive possible code from the class diagram of [figure 21](#). The collaboration model says more about the system and less about the code. Specifically, many classes could provide the objects of [figure 25](#). We can choose our class structure independent of the collaboration. We could, for example, let the objects [Bjarne](#), [Gina](#) and [Trygve](#) be implemented by the classes *Man*, *Woman* and *Man* respectively. Or they could all be implemented by class *Person*. We could probably need to add an attribute *sex* to class *Person* and introduce a constraint to prevent sexless reproduction.

A colon (:) in front of a name indicates a class name in a UML collaboration. If an object could have been implemented by different classes, their names are given in a comma separated list as indicated in [figure 26](#).

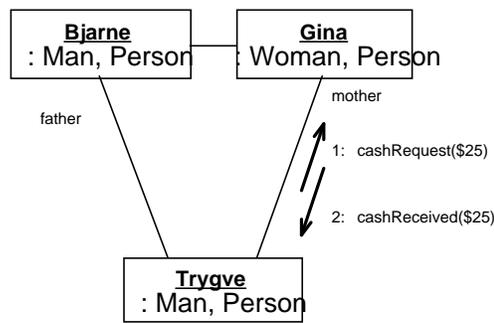


Figure 26. A simple interaction

The choice of classes for the different objects will, of course, strongly influence the details of the code.

Consider our activity network example shown in figure 4. *Frontloading* is an operation on the network that computes the earliest starting time for each activity given the start time for the project as a whole. Figure 27 shows the network.

We will now model the network as an open system as defined in section 2.3.2.

Try answering the following questions without looking ahead:

1. What is the purpose of the collaboration ?
2. What are the environment objects ?
3. What are the system objects involved in this operation?
4. What are the responsibilities of these objects?
5. What are the message links that connect them ?
6. How are messages flowing between the objects? Assume that the project starts in week 1.

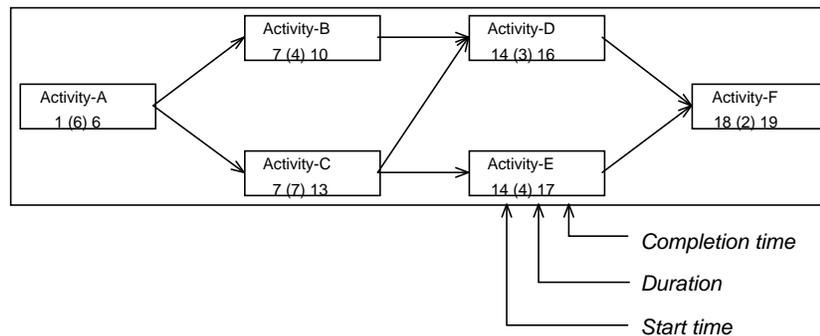


Figure 27. Exercise: How do the objects collaborate to perform frontloading?

Ponder the questions !
Ponder the questions !

What is the purpose of the collaboration ?

A few paragraphs ago, we stated that the purpose is *frontloading*. We can specify a use case for this operation: *UC1 frontloading*.

What are the environment objects ?

We return to our demo in [figure 5](#) and see that frontloading is triggered by the user pushing the *frontload* button. So the user acts as an environment object and could be modeled as an actor as shown in [figure 28](#) (a). There is a user interface between the user and the network model proper. Remember that we can choose what we want to include within the system boundaries. So we have two alternatives as illustrated in [figure 28](#). We can include the user interface within the system and let the user be an actor modeled as a Person object (a), or we can restrict the system to include the network model itself and let the user interface be the environment object (b).

The user interface is quite complex and consists of many objects. A fruitful separation of concerns is to specify the user interface as a system, separate from the system representing the activity model as such. So we choose alternative (b). In the following, we focus on the activity network system.

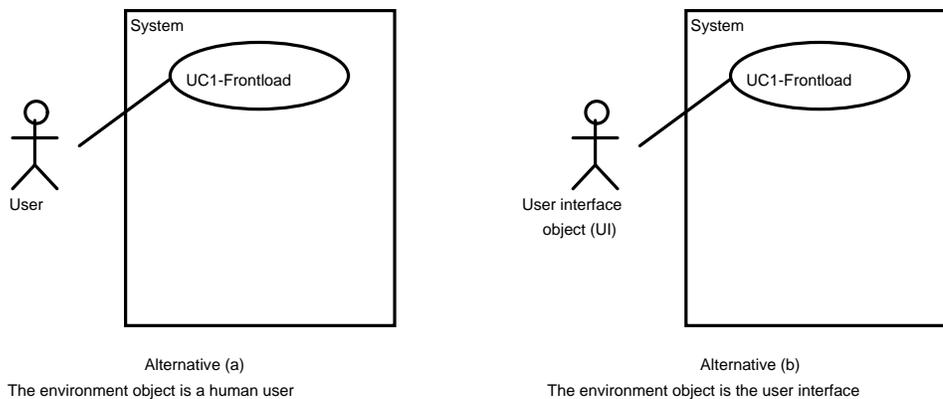


Figure 28. Alternative use case diagrams for the frontloading operation

What are the system objects involved in this operation?

It seems reasonable to represent each activity as an object. This supports our original plan for realizing the planning operation with a system of negotiating objects. If we only have activity objects, the UI must interact with all these objects. This solution leads to a very tight coupling between the network model and the user interface since the UI must reference all the activities. We much prefer a simpler coupling, and introduce a kind of master object; let's call it *Project* since it represents the whole.

What are the responsibilities of the objects?

We couldn't answer the previous question without answering this one in our mind.

1. The *user interface (UI)* object is responsible for bridging the gap between the user and the network model.
2. Each *activity* object is responsible for scheduling an activity in cooperation with the object's predecessor and successor objects.
3. The *project* object is responsible for managing the activity objects and for the overall control of the frontloading operation and represent the network towards its environment.

What are the message links that connect them ?

How are messages flowing between the objects? Assume that the project starts in week 1.

We answer these questions together because we need links where we need to pass messages. The collaboration and interaction diagram of [figure 29](#) shows the message flow

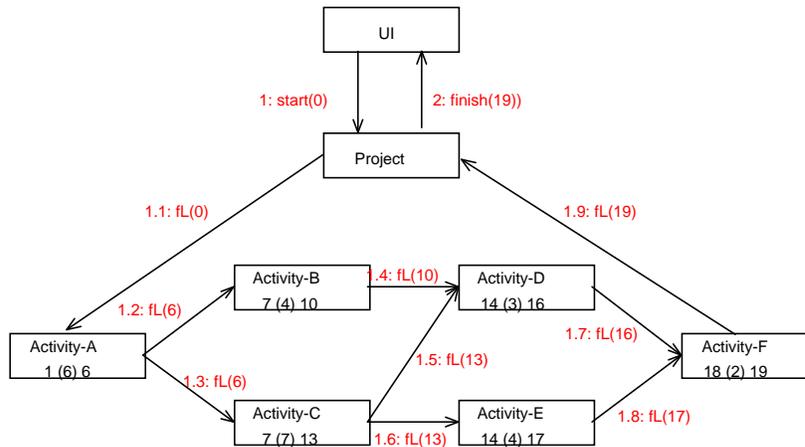


Figure 29. The example Activity Network: Objects and Interaction
fL(t) means *frontLoad(int time)*;

Figure 29 shows the corresponding message sequence chart. The objects are in the top row. Time flows down from the objects. Messages are shown as horizontal arrows, each arrow indicating that a message is sent from the arrow start to the arrowhead.

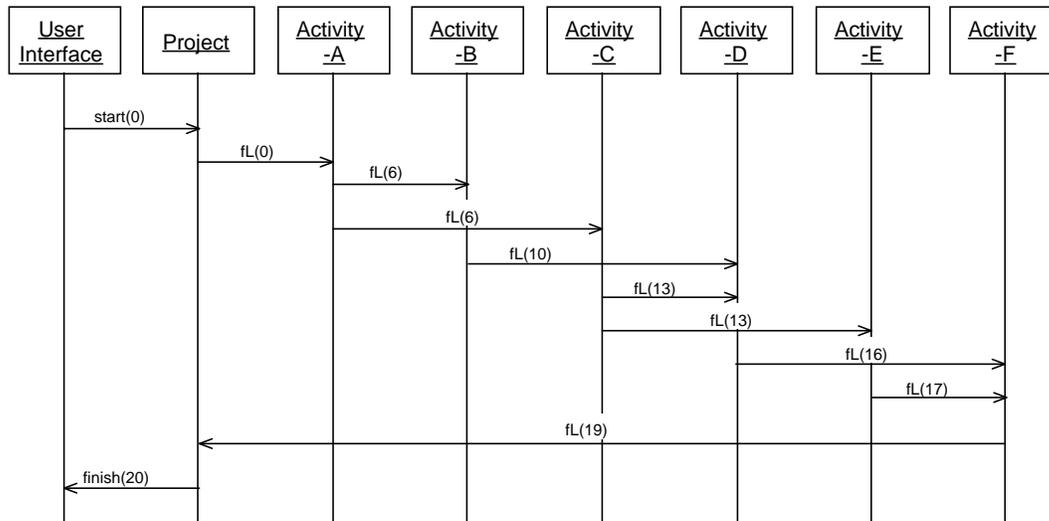


Figure 30. The example Activity Network: Message sequence chart

< ? much more about message sequence chart somewhere ? >

We know a great deal about the system and how it functions. But what do we know about the code? A great deal, actually. But not the same as for the class model:

Each of the 8 objects has to be an instance of some class. It would be easy to write a common class for all objects. Or three classes, for example *UI*, *Project* and *Activity*. Or each object could have its own class. This last choice could make sense if the details of those classes were different. They could have a common superclass if they had some code in common.

Let's look at each object in turn and discuss what we know about its class:

1. UI. This object is probably part of a cluster that makes up the user interface. It must have a method that sends the *start()* message. This method is to be triggered by some event in the environment. The UI must also have a *finish()* method that receives the final message from the project.
2. Project. The class must have a *start()* method that sends the *frontLoad()* message to the start activities. It must also implement a *frontLoad()* method that receives the final *frontLoad()* messages from the end activities. This method must cause the *finish()* message to be sent to the UI.
3. Activity-A through Activity-F. All activity classes must implement a *frontLoad()* method. This method must cause the *frontLoad()* message to be sent to all successors when it has received the *frontLoad()* message from all its predecessors. It must also have sufficient information for it to compute its completion time from its start time, e.g., its duration.

So the class structure of [figure 31](#) is a reasonable first solution. We define a separate *Project* class because its methods shall be different from all the others. We define a single *Activity* class because there is nothing in the collaboration that indicates that they need be different. We add attributes for the interesting planning data, *earlyStart*, *earlyFinish*, and *duration*. The two first are of interest to all objects, while the *duration* is something we need to know in the activities in order to compute their finish-times. We do not show a class *UserInterface* since this is outside the system under consideration.

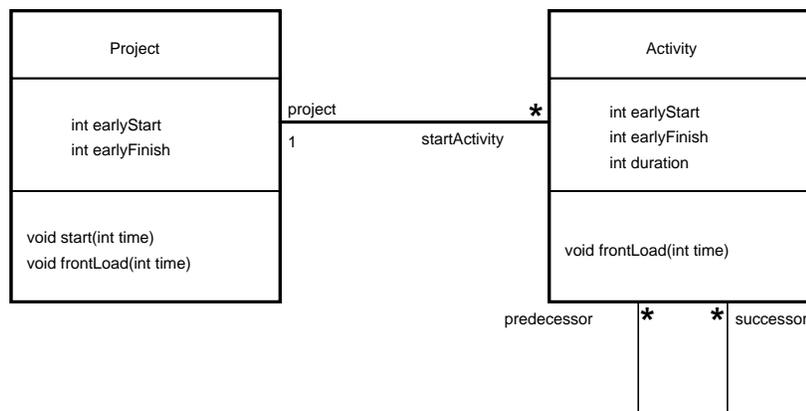


Figure 31. Possible class model for a simple activity network implementation

We first write some skeleton code for class *Project* based on the information in the collaboration model:

```

public class Project {
    /* start activities */
    public Activity[] startActivities;
    /* project start week */
    int earlyStart;
    /* project completion week */
    int earlyFinish;
    /* The actor (an environment object) sends the following message to start the
    operation */
    public int start(int time) {
        // must send frontLoad() to all startActivities
        // let it return the project early finish time
        // so that the project does not have to know its client
    }
    /* predecessor reports week of completion */
  
```

```

    /* Real project finish is the maximum from all end-activities */
    public void frontLoad(int time) {
        // This method must report the project finish time to the actor object.
    }
}

```

We also write skeleton code for class *Activity* that conforms to the collaboration with one exception: There is a serious typing problem with this code. Can you see it?

```

public class Activity {
    /* activity start week */
    int earlyStart;
    /* activity completion week */
    int earlyFinish;
    /* activity duration in weeks */
    int duration;
    /* predecessor reports week of completion */
    public void frontLoad(int time) {
        // This method must collect info from all predecessors
        // compute the completion week,
        // and report it to all successors.
    }
}

```

The problem is that the type given for *successors* in class *Activity* is *Activity*. The system interaction shown in figure 29 and figure 30 specify that the last *frontLoad()* messages shall go to the *Project*. So there is a type error. This is exactly the problem that stumped us 25 years ago. Simula insist that we know the class of the objects that are referenced by a variable. There are ways around it. We can, for example, introduce an artificial superclass or we can type the variable to *Object* and use type casting.

But these and solutions are obfuscating and inelegant. A much better solution is to say exactly what we mean. Type the variable to an *interface* that specify the exact set of messages that we permit to be sent out of that reference, i.e., type the out basket of the clerk metaphor. (figure 13). Interfaces are first class citizens of the Java language, and experts tell me that the same effect can be achieved in C++ with abstract superclasses. We will return to this solution in [section 5](#) where we type the references (association end roles).

Let's compare the choices made for the class and collaboration abstractions:

What's important in the class abstraction	<i>What's important in the collaboration abstraction</i>
Objects are encapsulations of data and behavior.	<i>An object is an encapsulation of data and behavior.</i>
Objects are instances of classes.	<i>An object has a unique identity.</i>
Associations between two or more classes specify links between their instances.	<i>An object collaborates with another object by message interaction.</i>
An operation describes a service that can be requested from an object to effect behavior.	<i>An object's visible behavior consists its typed input and output links and interfaces.</i>

To give an oversimplified and somewhat vulgar summary: UML offers two principal and complimentary abstractions on objects. The *class abstraction* models system construction by focusing on information structure and object features. The *collaboration abstraction* models system behavior by focusing on patterns of interacting objects. Both provide partial specification of the classes; neither provides a complete specification. (But the UML class abstraction includes a great deal of optional information that is pertinent for some popular programming languages. The UML collaboration is much leaner in this respect.)

4 The specification level Collaboration

The many collaborations of the previous chapter had one thing in common; they were *instance level collaborations* because they described systems by describing their actual objects. Such collaborations can be very illuminating for describing the architecture and behavior of simple object structures. But they are too specific for more complex systems. For example, strictly interpreted the family model of [figure 26](#) applies to me and my mother only. We would need a new model to show how my brother can get money from our mother.

The *specification level collaboration* lifts the power of describing system behavior to a higher level. We generalize the idea of a collaboration by representing the objects by *ClassifierRoles*, naming them by the name of the *responsibility* or *position* they hold in the object system; we call it the *role* they play in the system.

position [Webster]

4a: the point or area occupied by a physical object <took her position at the head of the line>
5a: relative place, situation, or standing <is now in a position to make important decisions on his own>

role [Webster]

1a (1) : a character assigned or assumed
(2) : a socially expected behavior pattern usually determined by an individual's status in a particular society
2 : a function or part performed especially in a particular operation or process

In [figure 32](#), we have generalized our simple clerk metaphor of [figure 13](#). Think of it as a pattern or rubber stamp describing how a group of objects work together for a certain purpose. [Figure 23](#) shows an instance of this collaboration; other groups of compatible objects could enact the same collaboration.

The notation is similar to the instance level collaboration notation. The only difference is that the names start with a solidus ('/') rather than being underlined. Many people would prefer clearer symbolic differences. It is particularly easy to confuse the rectangular class symbol in the class diagram with the rectangular role symbol in the collaboration diagram. But this is how it is.

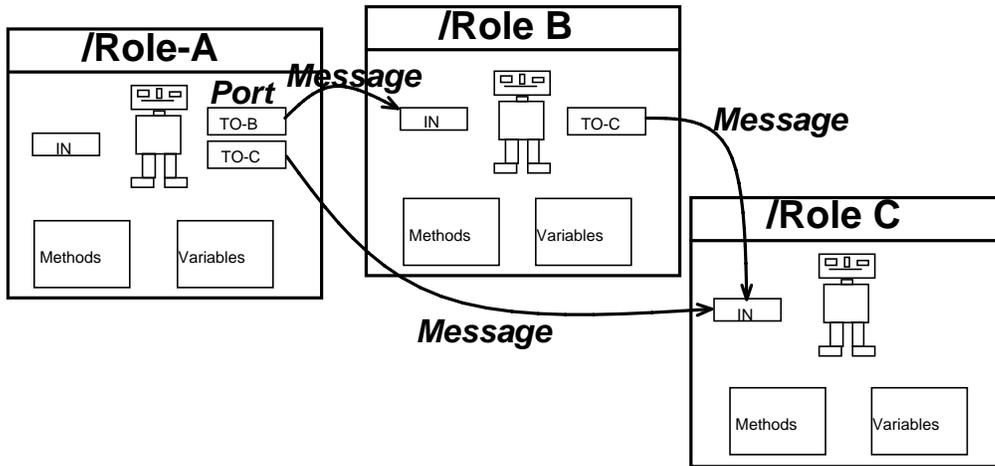


Figure 32. The role playing metaphor

The idea is to search out patterns of interacting objects based on the *role* the objects play in the interaction. For example, objects in my small family play three roles: /Mother, /Father and /Child as illustrated in [figure 33](#).

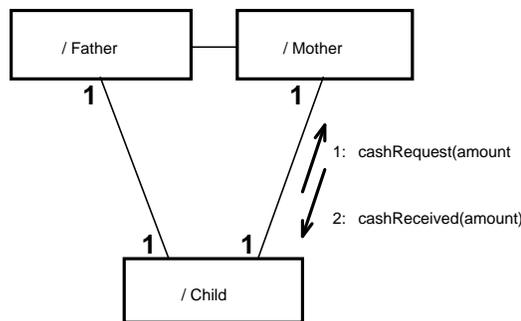


Figure 33. Specification level collaboration a simple example

The message sequence chart is still a good notation for describing system behavior as you can see from [figure 34](#).

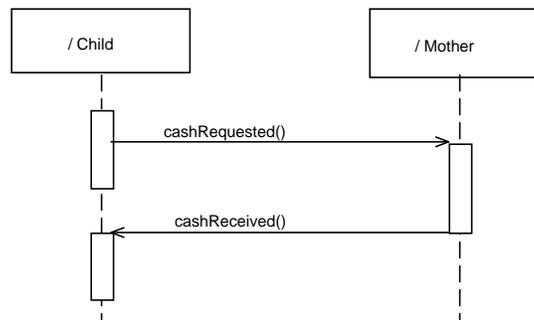


Figure 34. A message sequence diagram showing a specification level interaction

In the specification level collaboration, every object is represented by a slot that can hold an object; it names the role that the object plays when it occupies this position. The slots are called

ClassifierRoles, they form a pattern that can be instantiated again and again: Even at my ripe old age I play the role of /Child in relation to my /Mother and /Father. I also play the role of /Father in relation to my daughter, etc., etc.

The cardinality of the /Child role in [figure 33](#) is ONE. This does not mean that a father and mother has exactly one child. It only means that for the purposes of this particular collaboration, we single out a particular child for consideration.

The definition of a Collaboration applies equally to the instance and specification levels:

Collaboration

A Collaboration describes how a number of objects work together for a common purpose. There are two aspects. The structural aspect is a description of the responsibilities of each object in the context of the overall purpose of the collaboration; and also the links that connect the objects into a communicating whole. The dynamic aspect is a description of how stimuli flow between the objects to achieve the common purpose.

We also need to define how the model elements on the specification level relate to the elements on the instance level:

ClassifierRole

A named slot for an object participating in a Collaboration. Object behavior is represented by its participation in the overall behavior of the Collaboration. Object identity is preserved through this constraint: "In an instance of a collaboration, each ClassifierRole maps onto at most one object."

Note that the constraint is asymmetrical. An object can play several roles in one or more Collaborations, while a role maps onto a single object in an instance of the collaboration.

Also note that the parts have very little meaning when isolated from the whole. Indeed, in UML 1.3 the ClassifierRoles form a composition aggregate under the Collaboration; they are considered meaningless outside this context.

Instantiating class *Man* gives an instance of *Man*. In contrast, it is meaningless to instantiate a ClassifierRole in isolation. The role gets its meaning from its position in the Collaboration and can only be instantiated in this context. There can be no child without a father, and there can be no mother without a child. For example, instantiating the /Father role involves the following:

1. Select a suitable factory object.
2. Ask the factory object for an instance that has the features needed to play the role.
3. Connect the resulting object into the collaboration structure. For example, this may involve linking the father object to the child and mother objects, as well as linking the mother and child objects to the father object.

Let's return to the activity network example. The question we have to ask ourselves is: "What is the essence of the frontloading operation as it is performed by the system of objects?" The object interaction diagram is in [figure 29](#). We find an avalanche of messages passing successively from predecessor to successor. The specification level collaboration of [figure 35](#) seems to capture the essence.

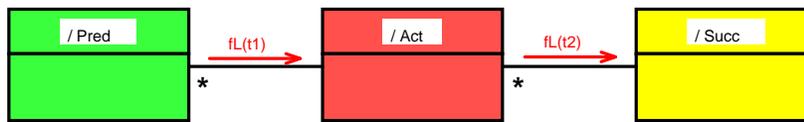


Figure 35. Specification level Collaboration. Frontloading Use Case

The message sequence chart of figure 36 is an alternative notation for the same interaction.

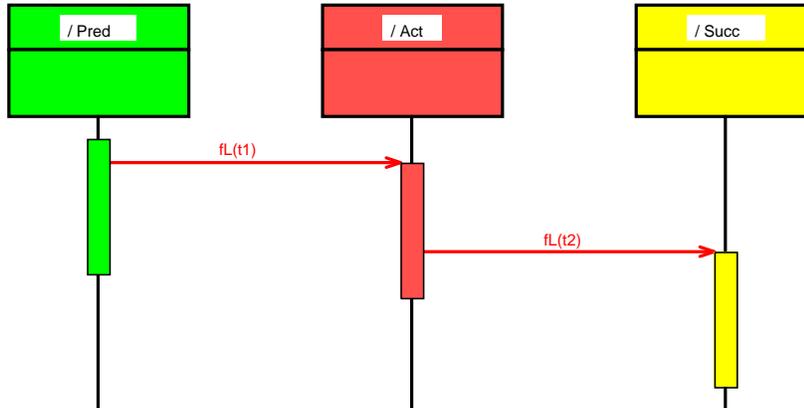


Figure 36. Specification level Collaboration. Frontloading Sequence Diagram

You may wonder if the cardinality MANY in figure 35 is inconsistent because of the rule "In an instance of a collaboration, each classifier role maps onto at most one object". We add the following rule to make it all nice and consistent: "a MANY relation means that the role represents a typical member of the set of objects. All other members of the set behave correspondingly".

We can construct the object structure of the activity network example in figure 29 by combining six instances of this specification level collaboration as given in the table below. The same information is illustrated graphically in figure 37.

Instance	ClassifierRole		
	/Pred	/Act	/Succ
#1	Project	Activity-A	Activity-B Activity-C
#2	Activity-A	Activity-B	Activity-D
#3	Activity-A	Activity-C	Activity-D Activity-E
#4	Activity-B Activity-C	Activity-D	Activity-F
#5	Activity-C	Activity-E	Activity-F
#6	Activity-D Activity-E	Activity-F	Project

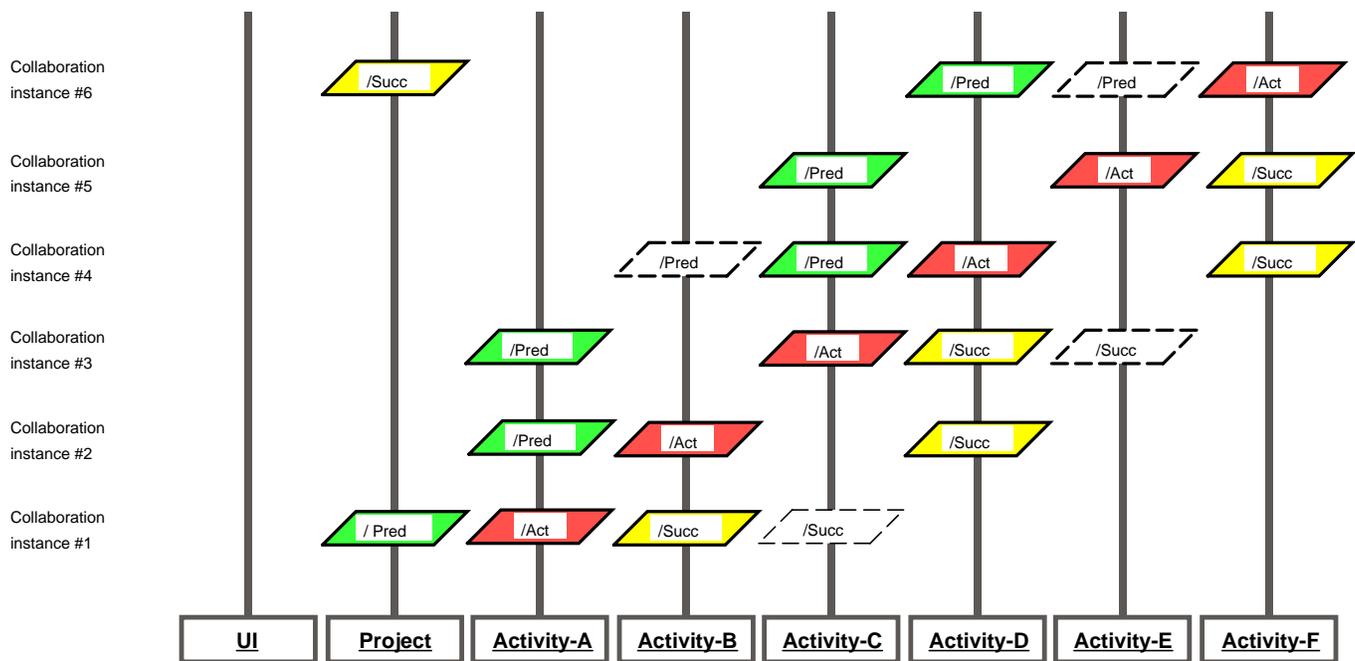


Figure 37. Our example network constructed from six instances of the collaboration model.

Notes:

1. The project and activity objects are almost indistinguishable in this context. In instance #1, the project is a predecessor and in instance #6, it is a successor.
2. Activity-A and Activity-C have several successors. Since a classifierRole maps onto a single object, we elect one of them to represent the role. By convention, its peers are constrained to follow the selected one like a shadow. So in collaboration instance #1, Activity-C shall receive exactly the same messages as Activity-B and at the same time. The same applies to Activity-E and Activity-D in instance #3. Finally in instance #5, Activity-D is arbitrarily designated as instantiating /Pred while Activity-E must behave correspondingly. So by these conventions, Activity-E has to wait for frontLoad messages from *all* predecessors before it can proceed to the next step of the frontloading process.

Closer study of figure 37 shows that the UI object doesn't play any of the roles. Our collaboration is clearly not a complete realization of the Frontloading use case. Deeper thought reveals that the management of the frontloading operation is not covered by our collaboration model. This management is an interaction between the UI and the Project; and outside the scope of the above frontloading collaboration.

The solution is to apply separation of concerns. We subdivide the use case into a management part and a computation part as follows:

UC 1: Frontloading

1. *UC 1.1: Frontloading management*
2. *UC 1.2: Frontloading computation*

Our current collaboration resolves UC 1.2. We add another collaboration for UC 1.1 as shown in

figure 38.

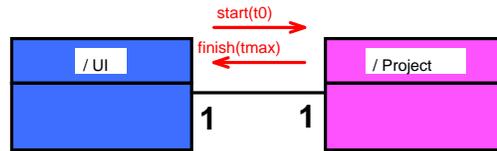


Figure 38. UC 1.1: Frontloading management collaboration

We now construct the complete frontloading interaction (figure 29) from these two collaborations. The result is shown in figure 39.

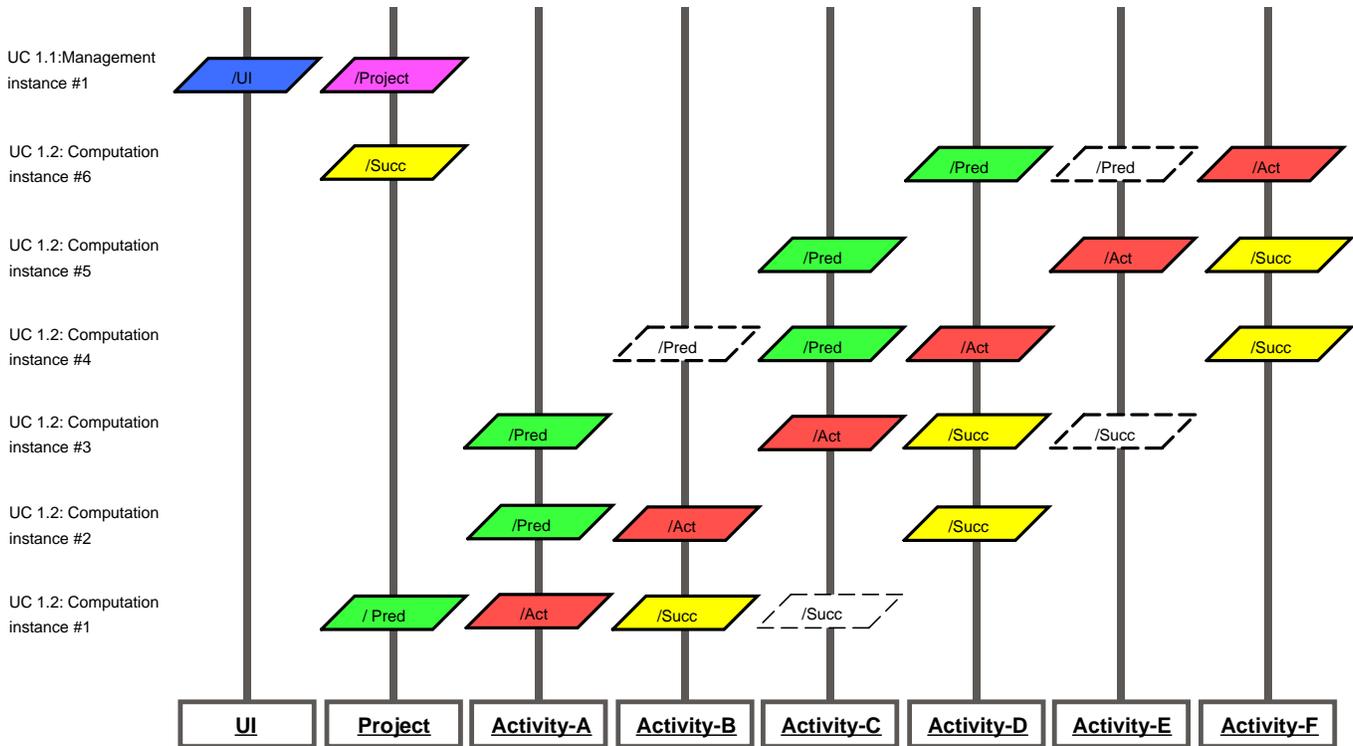


Figure 39. Our example network constructed from six instances of the collaboration model and one instance of the frontloading management model. Each collaboration

<Probably want to add another sub-UC: reset frontloading to prepare for the computation. >

Collaboration literacy entails being able to envisage the objects of any activity network as being constructed from the simple collaboration models of figure 35 and figure 38 and likewise to envisage how its system behavior is constructed from the behavior of each of the collaboration models. Once a decision has been made as to which classes shall implement which roles, the coding should be straight forward.

Let's deduct the code of the previous chapter from our two collaborations. We first assign classes to the roles as shown in figure 40.

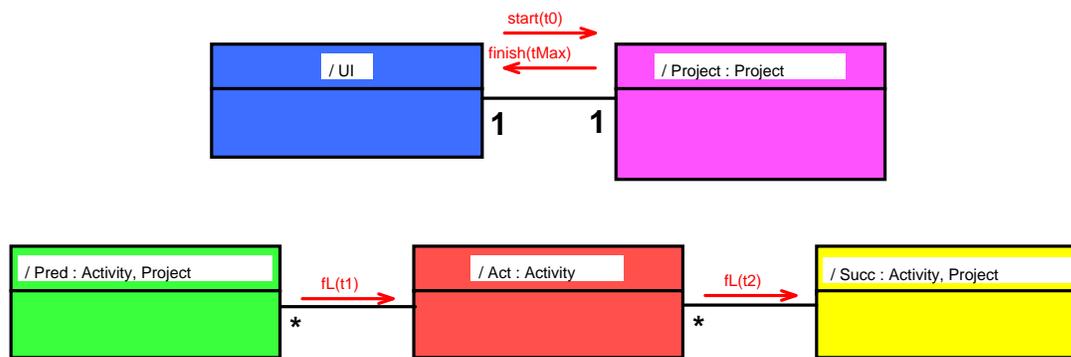


Figure 40. Assigning classes to the roles

So what does all of this mean in terms of code? Exactly the same as for the instance level collaboration. We have used separation of concerns to partition, generalize and simplify the models. But the semantics is unchanged and the code can be identical!

A ClassifierRole may be thought of as a slice of a Class. But this is an oversimplification; it ignores its essential relationship to the Collaboration as a whole. A better metaphor is to think of a ClassifierRole as specifying a slice of an object; namely the AttributeLinks, Links, and behavior that it needs as a participant in the collaboration.

Can you identify the code that arises from the different roles in the following code?

```

public class Project {
  /** start activities */
  public Activity[] startActivities;
  /** project start week */
  int earlyStart;
  /** project completion week */
  int earlyFinish;
  /** The actor (an environment object) sends the following message to start the
  operation */
  public int start(int time) {
    // must send frontLoad() to all startActivities
    // let it return the project early finish time
    // so that the project does not have to know its client
  }
  /** Predecessor reports week of completion.
  Real project finish is the maximum completion from all end-activities */
  public void frontLoad(int time) {
    // This method must report the project completion time to the actor object.
  }
}

public class Activity {
  /** Activity start week */
  int earlyStart;
  /** Activity completion week */
  int earlyFinish;
  /** Activity duration in weeks */
  int duration;
  /** Predecessor reports week of completion */
  public void frontLoad(int time) {
    // This method must collect info from all predecessors
    // compute the completion week,
    // and report it to all successors.
  }
}

```

}}
}

We make the example somewhat more realistic by separating the end user tool from the network logic. We arbitrarily choose to implement the tool as a Java applet, and to implement the activity network as an Enterprise Java Bean (EJB). The collaboration diagrams are still valid. The communication path in the UC 1.1 collaboration is a remote connection; while the communication paths in the UC 1.2 collaboration are all local. The logic is the same; but we have to consider issues such as efficiency, security and exceptions to accommodate the differences in messaging mechanisms. The beauty is at all these issues can be analyzed within the context of the collaborations.

How does a user interface make use of an Enterprise Java Bean to provide the business logic for an application?

<Need to write more about this, probably in a separate chapter>

Figure 40 shows how a remote client can establish contact with a Java Enterprise Bean

<Rework for activity network example. Change color coding to conform with previous diagrams.>

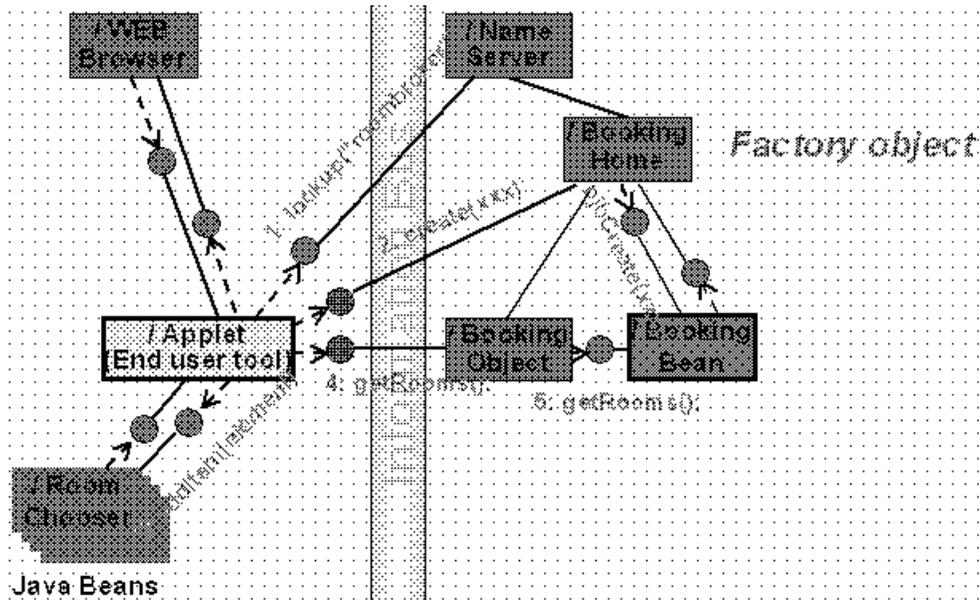


Figure 41. Enterprise Java Bean. Application Assembler perspective

Note how each object has responsibility, knows its collaborators and is robust. Nobody knows everything! The code is invisible at this level of description. Objects, references and interfaces are visible.

5 Typing the AssociationEndRoles

Let us again consider our metaphor for the specification level collaboration. We now want to specify the messages that a clerk shall be allowed to send to each of its collaborators. This gives much finer grained control than merely typing the *ClassifierRoles* themselves. We do it by attaching an Interface to each of the out baskets:

interface. A named set of operations that characterize the behavior of an element.

UML offers two alternative notations. [Figure 35](#) uses the generalization arrow, while [figure 36](#) uses the "lollipop" symbol that denotes an interface.

[Figure 35](#) and [figure 36](#) show the family cash transfer example with interfaces added to the specification level collaboration. This model is more specific than the class diagram in [figure 21](#) because it says that only a */Child* can ask for cash from its */Mother*. It also says that any class implementing the */Mother* role must understand *cashRequested()*. Likewise, any class implementing the */Child* role must understand *cashReceived()*. It could be *class Mother* or *class Person*, or even *class Benefactor*. Or should we do it the other way around so that our current model has two roles: Beggar and Benefactor. We can then later implement these roles onto various classes depending on the details.

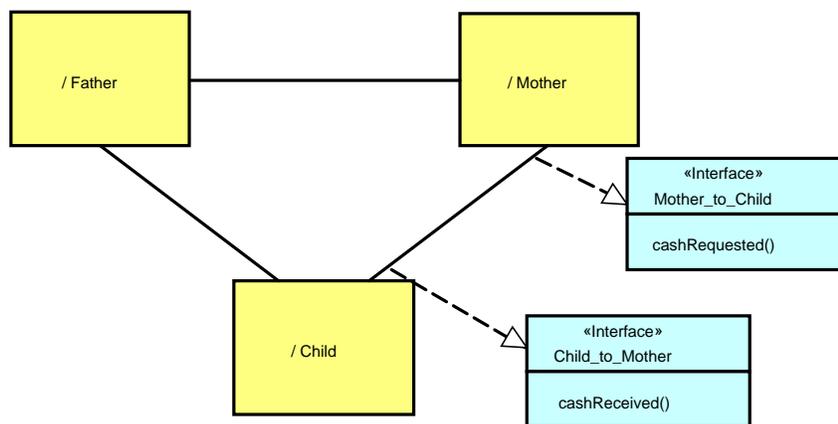


Figure 42. Typing the out baskets (AssociationEndRoles)

The practical consequence of this is that we may type the link from a *ClassifierRole* to a collaborator.

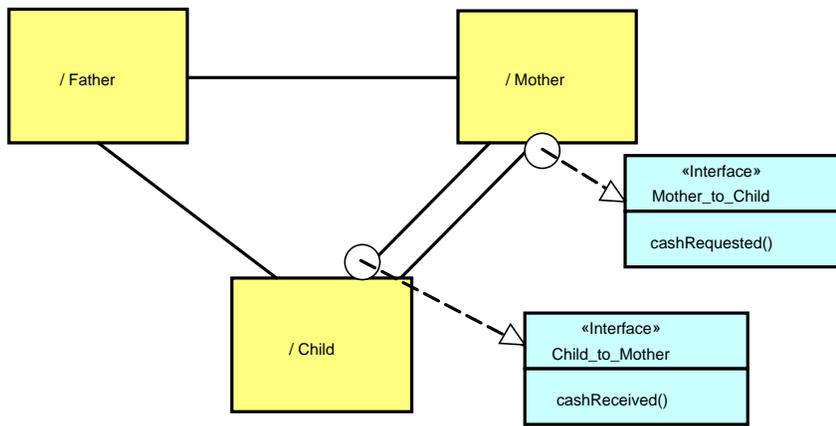


Figure 43. Showing interfaces as "lollipops"

The interfaces specify minimal requirements to receiving classes

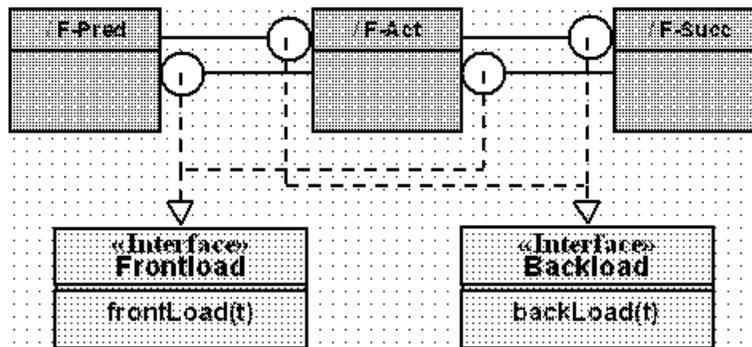


Figure 44. Interfaces in the basic scheduling collaboration

Defining these interfaces in Java:

```

public interface Frontload {
    public void frontLoad (int t) ;
}
public interface Backload {
    public void backLoad (int t) ;
}

```

Let's implement these roles with a *Project* and an *Activity* class as specified in the following table:

collaboration role	implemented by classes
F-Pred	Activity, Project
F-Act:	Activity
F-Succ	Activity, Project

A role can be implemented by one or more classes. This is optionally indicated in the collaboration diagram with a colon (:) preceding the list of class names as shown in [figure 28](#).

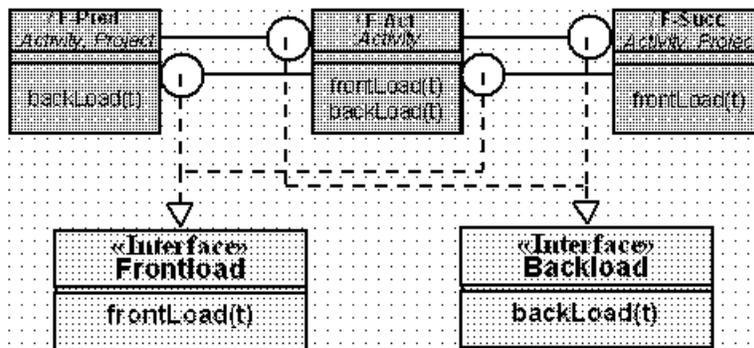


Figure 45. Possible implementations of the basic scheduling collaboration

The following class declarations satisfy the requirements:

```

public class Project
  implements Frontload, Backload{
}
public class Activity
  implements Frontload, Backload{
}
  
```

6 Specialization of a Collaboration

In UML 1.3, *Collaboration* is syntactically a specialization of *GeneralizableElement*. The specialization of a Collaboration is precisely defined in OOram. Other methods such as Catalysis define different mechanisms. So, in my opinion, UML should avoid making narrow definitions that will exclude useful variants. It is better to define a general mechanism and open for different specializations through different stereotypes.

I will briefly describe the OOram specialization operation, called *synthesis*. The MVC can be a suitable first example. There are three ClassifierRoles: The */Model* that is responsible for storing and managing some information. The */View* that is responsible for presenting this information to an external actor. The */Controller* that is responsible for taking commands from the same actor.

Let's add another Collaboration. We rephrase the Observer pattern [Gamma et. al.: Design Patterns] as a Collaboration with two Roles: */Subject* and */Observer*.

We can now specialize these two models into an ObserverMVC by specifying that a new OModel role shall play both the Model role and the Subject role. Further, the new OView role shall be required to play View and the Observer roles. There are *two* synthesis operations: First, the ObserverMVC is made a specialization of MVC. Second, the ObserverMVC is also made a specialization of the Observer model.

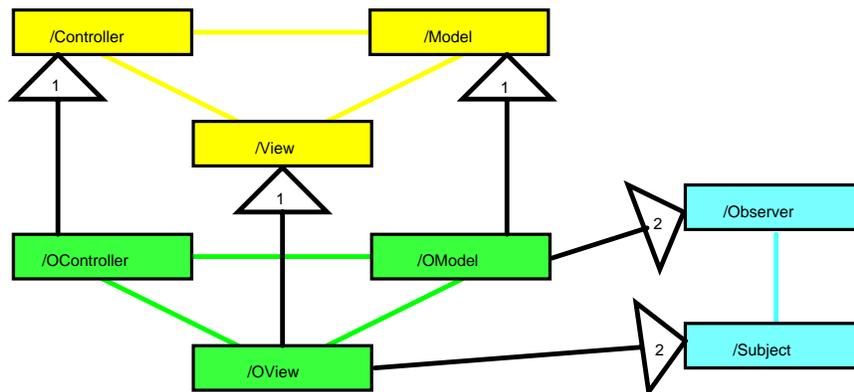


Figure 46. The ObserverMVC collaboration specified by synthesizing the MVC collaboration with the Observer pattern

Notice that it is meaningless to specialize a single ClassifierRole. If we only specialize */View* into */OView*, it will never receive an *update()* message. Similarly, if we only specialize */Model* into */OModel*, there will never be any observers.

There is a truism here: If we specialize a class by adding new features, we must also specialize some (other) class to utilize these new features. The argument is even stronger when specializing a Collaboration: If we want to retain not only the features of the individual Roles, but also the overall behavior of the Collaboration, then we must map all the roles of the parent model onto roles of the child model.

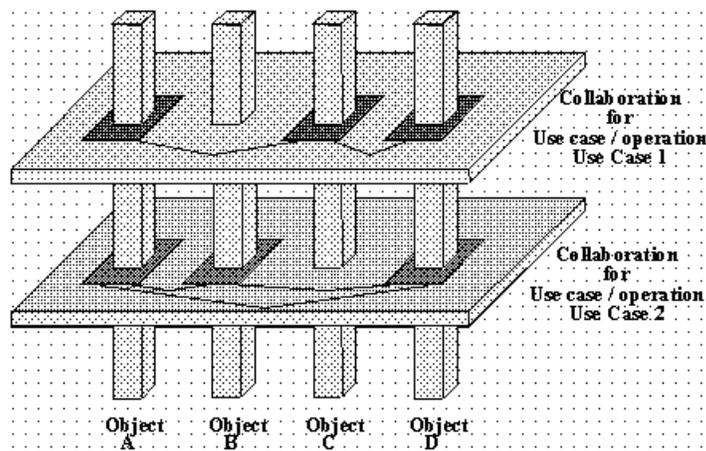


Figure 47. Generalization of Collaborations; OOAM Synthesis

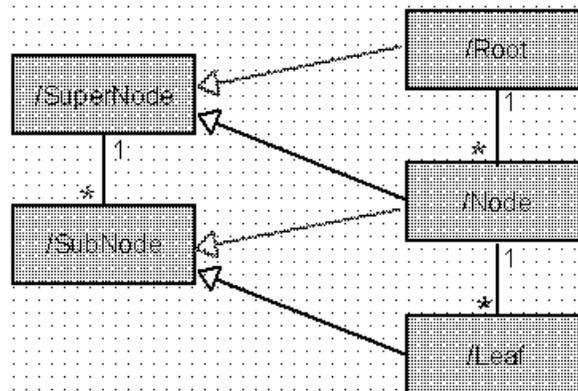


Figure 48. Example illustrating the definition: Create three level tree from basic tree

Exercise: Activity planning with resources

Planning with resources. Use Cases:

- ☒ Frontloading
- ☒ Resource allocation for single activity

For last use case:

- ☒ What are the objects ?
- ☒ What are their responsibilities ?
- ☒ How are they interconnected ?
- ☒ How do they interact ?

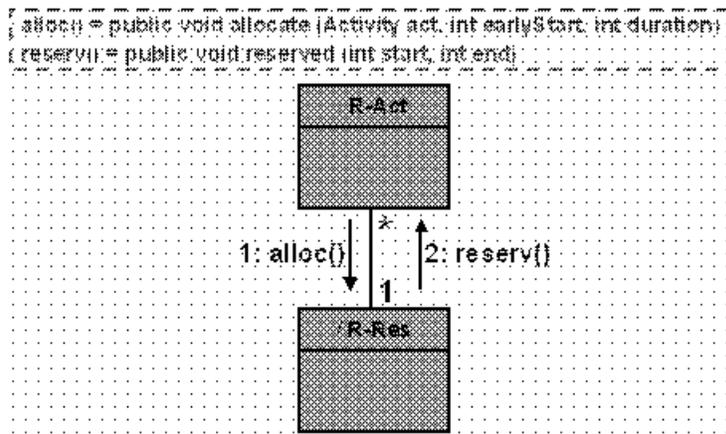


Figure 49. Activity planning: Resource Allocation Use Case

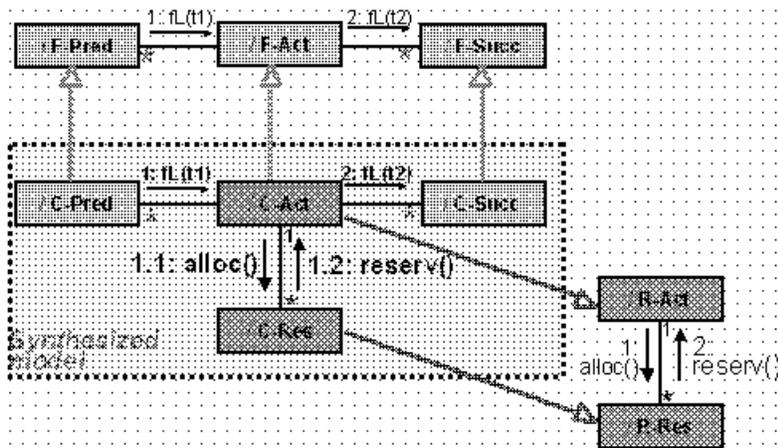


Figure 50. Specialization of Collaboration: Frontloading with resources

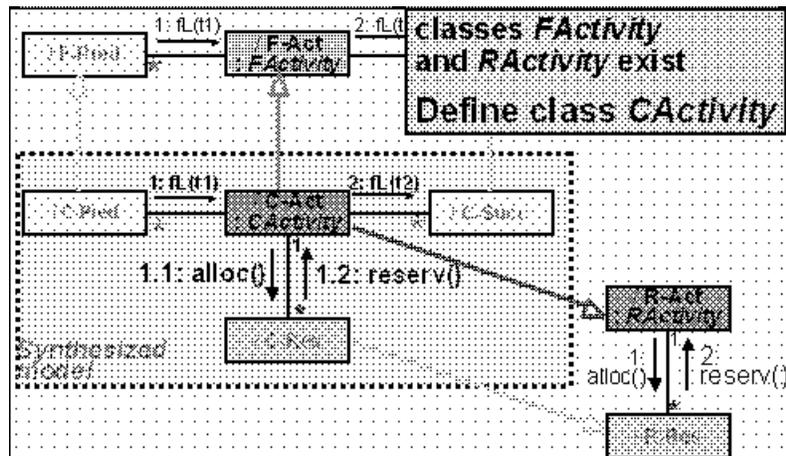


Figure 51. Exercise: Specialization of Classes

Specialization of collaboration often leads to multiple inheritance. This has to be resolved when implementing in single inheritance languages such as Java. For example:

```

public class FActivity
    implements Frontload, Backload

```

```

{...}
public class RActivity
  implements Reserved
{...}

public class CActivity
  extends FActivity
  implements Reserved
{...}

```

6.1 MVC example. note dated 17 June 1997

author: Trygve Reenskaug

receiver: UML Semantics

date: 17 June 1997

Old stuff to be merged into book.

The goal of this example is to explore the use of UML 1.1 for the definition and application of frameworks. I have based the work on what appears to be the latest semantic model: *beta-r1.doc*.

The example framework is the Model-View-Controller (MVC), the example application is a rudimentary Text Editor. For modeling, I have used some ad hoc forms of Use Case diagrams, Context diagrams, Scenarios, and Class diagrams.

I have used the following terms with a different meaning from the beta glossary:

1. **collaboration**

An abstraction of a system that describes the static and dynamic properties of a structure of interacting objects in the context of a given purpose. The structural component is called a context and the behavioral component is called an interaction (*alternatively*, the behavioral component consists of any number of interactions).

2. **interaction**

A behavioral specification that comprises a set of message exchanges among a set of objects within a particular context to accomplish a specific purpose. An interaction may be specified by a society of state machines, it may be defined by a set of constraints, and it may be illustrated by one or more scenarios.

3. **pattern**

A UML pattern is a generic collaboration intended for refinement.

4. **framework**

A UML framework is a UML pattern augmented by a corresponding society of classes intended for specialization. *{specialization is missing from the glossary of beta-r1.doc}*

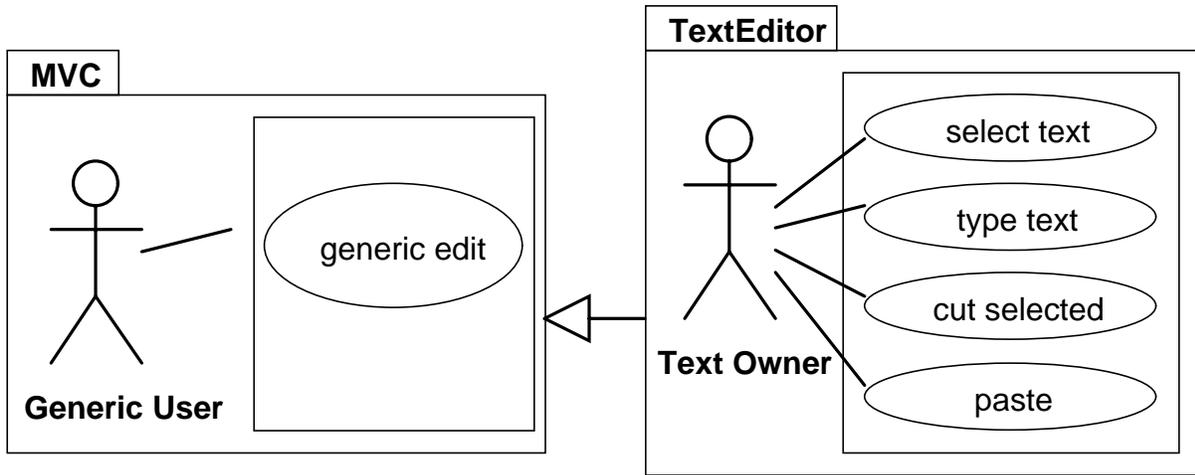


Figure 52. Use Case diagrams

COMMENTS:

- A) I have used the generalization symbol for refinement, this could be a bad idea. The use case refinement is distinct from the collaboration refinement, which again is distinct from the class specialization. But taken together, they could also be considered to form a higher level refinement.
- B) I have enclosed the use cases in packages. The idea is to indicate that the refinement applies concurrently to the whole construct, rather than independently to each component. The indicated packages will also include collaborations, interfaces, and classes.
- C) There is one MVC use case: *generic edit*. It are refined into four use cases in the Text Editor: *select text*, *type text*, *cut selected*, and *paste*. Such fan-out from pattern model to derived model is not in the current OOram and will not be part of UML 1.1.

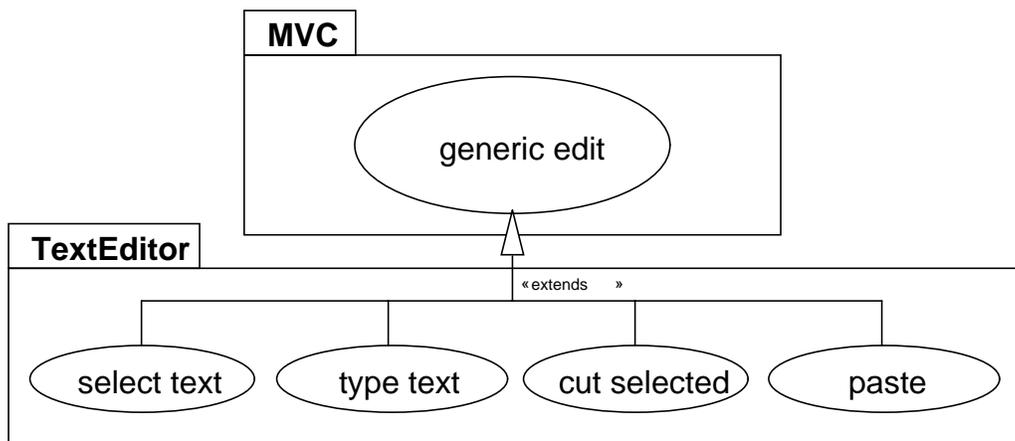


Figure 53. Use Case relationships

COMMENT:

A) This is an alternative notation for use case refinement found in UML 1.0. It conveys much the same information as figure 1, I am not sure if the package notation is permitted here.

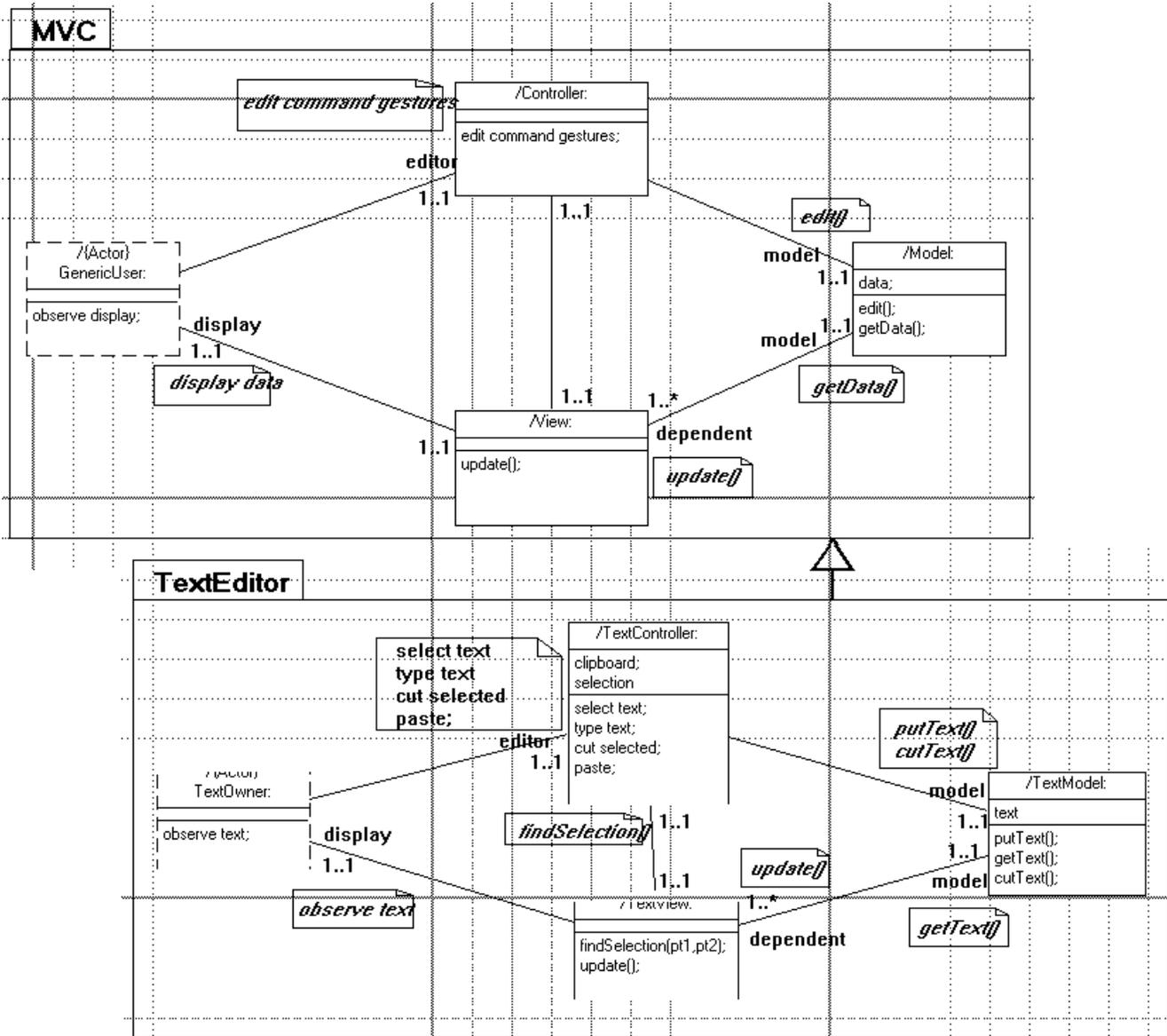


Figure 54. Context Diagrams

COMMENTS:

- A) The Actor slots are shown with dashed outlines. This is the OOram notation for environment roles, i.e., roles that either are the sources of interaction trigger messages or the sinks in the system environment that receive result messages.
- B) The interface of a *ClassifierSlot* is the union of the interfaces of its *AssociationRoleSlots*. The *ClassifierSlot* specifies the operations that need to be implemented, while the *AssociationRoleSlots* specify where they are used. For example, the *TextView* accepts *findSelection(...)* from the *Controller* and *update()* from the *Model*.

- C) I have enclosed the collaborations in packages. The idea is to indicate that the refinement applies concurrently to the whole construct, rather than independently to each component.
- D) I have used the generalization symbol for refinement, this could be a bad idea.
- E) I have indicated the *AssociationRoleSlot* interfaces with comment symbols.
- F) The generic *getData()* operation has been specialized into a *getText()* operation.
- G) The generic *edit()* operation has been specialized into two text editing operations.
- H) Refinement of framework operation signatures is in the current OOram, but is not included of UML 1.1 (??).

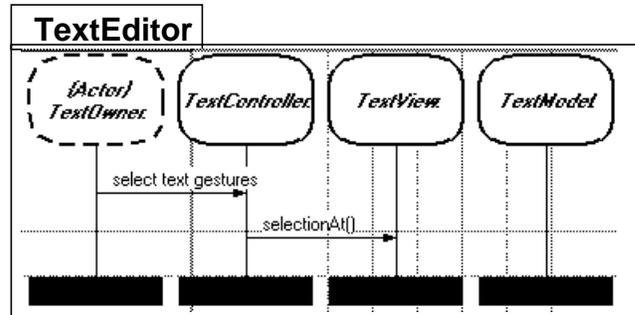


Figure 55. Scenario: "select text"

COMMENT:

- A) Selection is not defined as an MVC use case. Text selection is a TextEditor addition.

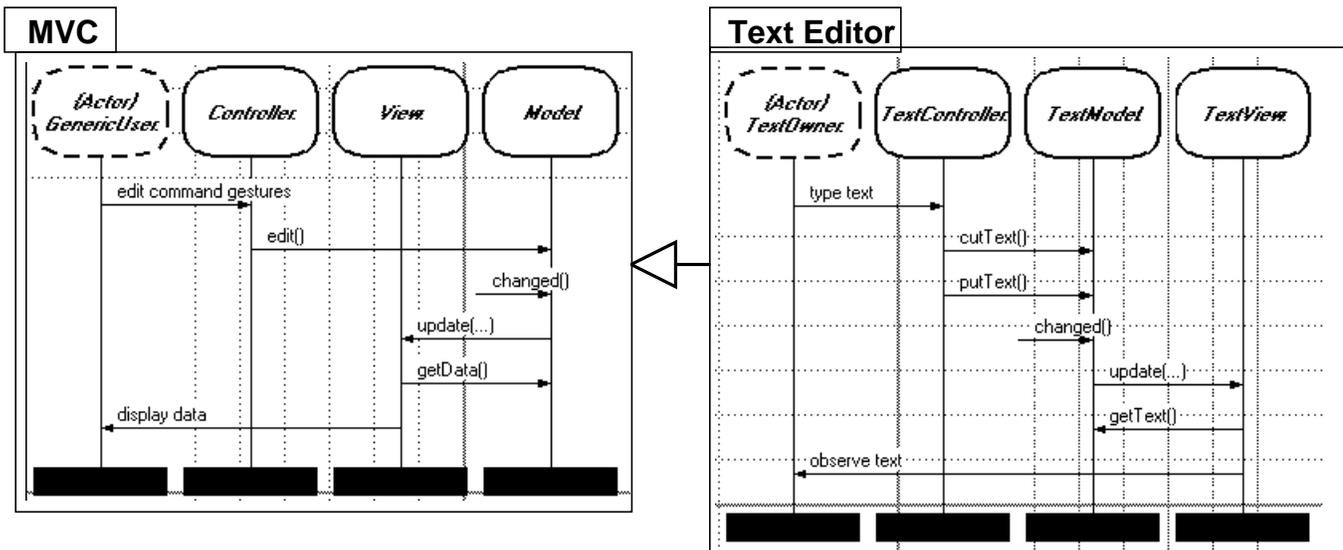


Figure 56. Scenarios: "generic edit" and "type text"

COMMENTS:

- A) The MVC *edit()* operation has been refined into a sequence of two TextEditor operations (*cutText(...)* and *putText(...)*).

- B) I have enclosed the Scenarios in the MVC/TextEdit packages.
- C) I have used the generalization symbol for refinement of Interactions, this could be a bad idea.

Figure 57. Class Diagram

COMMENTS:

- A) The similarity between the context diagram of figure 3 and the class diagram of figure 6 is only superficial; the *meaning* of the two diagrams are very different. A rectangle in figure 3 represents an object in a certain position within a pattern of interacting objects, while a rectangle in figure 6 represents a set of objects sharing the same properties. ***We deplore the use of rectangles for both concepts, and advocate that some alternative symbol be defined for the ClassifierSlot. OOram uses the somewhat exotic superellipse, may I suggest a rectangle with beveled corners as an presentation option?***
- B) The "type text" use case could be illustrated by additional Scenarios and other behavior descriptions.

The "cut selected" and "paste" use cases are similar and will not be shown here.

6.1.2 Metamodel considerations

I challenge the Architecture Work Group to check that the information expressed in the example diagrams can indeed be expressed through the metamodel constructs.

6.1.21 Re: figure 1: Use Case Diagrams

1. I believe Actor should be a ClassifierSlot, or a specialization of it. See figure 3. (In OOram, an Actor would be modeled as an environment role)
2. A Use Case represents some notion as a whole, there is no corresponding metaconcept. I have used Package, but a Package contains many other things beside use cases.
3. The abstract use case syntax has only MANY-MANY relationships that makes it hard to pin down a specific use case. I would consider making a UseCase contain Actors and Interactions (requirement holders and solutions). Then MANY-MANY between Actor and Interaction. Remove Interface from this abstract syntax.
4. Each of the elements of UseCase are generalizable. But the UseCase as a whole needs to be refined, the parts cannot be specialized independently of each other.

6.1.22 Re: figure 2: Use Case Relationships

See above.

6.1.23 Re: figure 3: Context Diagrams

The whole is a Collaboration, the parts are ClassifierSlots and AssociationSlots with their

AssociationRoleSlots. The semantics of the contexts in figure 3 seems to be covered. Figure 7 below is constructed from the abstract syntax of Collaboration, Core Relationships, and Core Backbone.

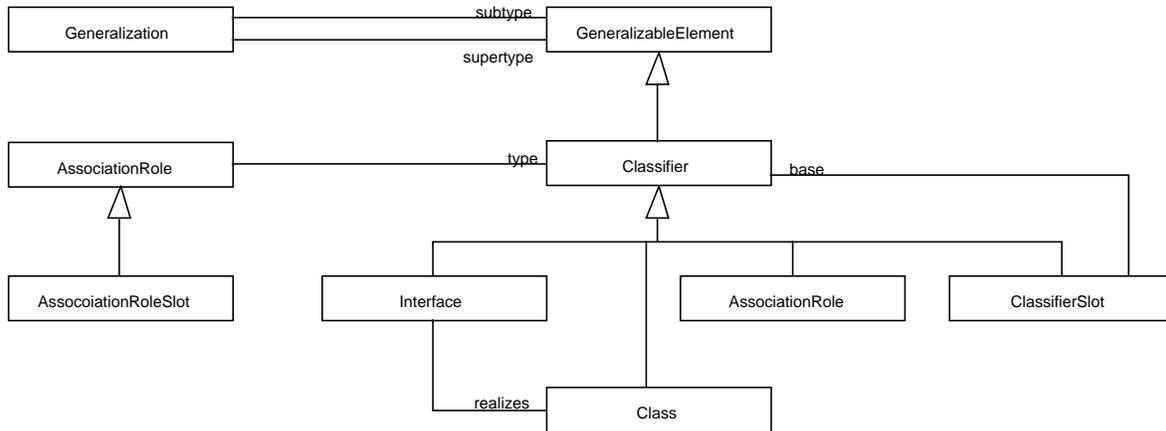


Figure 58. Composite abstract semantics that covers some OOram concepts

We see that the *type* of an AssociationRoleSlot can be an Interface. The Generalization dependency lets an Interface be *specialized* from several Interfaces. The base of a ClassifierSlot can be such a *specialized* Interface, or it can be a Class that *realizes* this Interface. The conclusion is that the basic OOram collaboration is covered in UML 1.1.

Comments:

1. The collaboration terminology is getting out of hand. These are concepts and terms that are to be used in the daily work of practical modelers. I suggest:
 - ClassifierSlot -> **Slot**
 - AssociationSlot -> **Link**
 - AssociationRoleSlot -> **Port**
2. Collaboration needs to be made into specialization of Namespace to make refinement work. (This also fits with the OOram role model, which is a name space)

6.1.24Re: figure 4: Scenario "select text"

Comments:

1. There is an anomaly in the UML definitions. Collaboration behavior can be defined in many different ways. A Collaboration can also realize many triggers (e.g., Use Cases).
 - The glossary says that an Interaction can be illustrated by one or more scenarios, the collaboration abstract syntax makes Interaction look like a single scenario.
 - Interactions may also be defined by state machines, possibly also constraints (Catalysis?). The abstract syntax does not permit this.
 - Maybe Behavior should be introduced between the Collaboration and the Interactions? Maybe there should be a clear distinction between Interaction (= algorithm for accomplishing a task) and Scenario (= example sequence of messages)

6.1.25 Re: figure 5: Scenarios: "generic edit" and "type text"

Comments:

1. Refinement of Interactions is a research issue and could well be left undefined in UML 1.1.
2. If we disregard the fan-out of a single MVC operation into several TextEditor operations, the refinement could be more precisely specified through Collaboration parameters and arguments.

6.1.26 Re: figure 6: Class diagrams

No comments.

7 System behavior, messages and interfaces

We should now have a clear picture of a Collaboration modeling a structure of interacting objects. But how do they interact?

UML offers two ways of modeling the avalanche of messages that flow from the sending objects to the receiving objects in a Collaboration: By Interactions or by State Machines. There is no magic; every message has a sender and a receiver. (Or, more precisely: The only magic is in the methods that are not first class citizens of a Collaboration model).

An example illustrates the importance of studying collaboration behavior. Consider the *instance level collaboration* shown to the right in the figure below. We can clearly associate a state diagram with each of the objects. If these diagrams show stimulus sends as well as stimulus receives, we can match the stimulus sends in one object with the corresponding stimulus receives in the receiving object. We can then check to see that the collaboration as a whole is consistent by checking that the receiving objects are in the appropriate state when receiving the stimuli, and that they will subsequently send expected stimuli to the other objects. (Notice that when considering the behavior of a collaboration, we ignore any behavior that is outside this context).

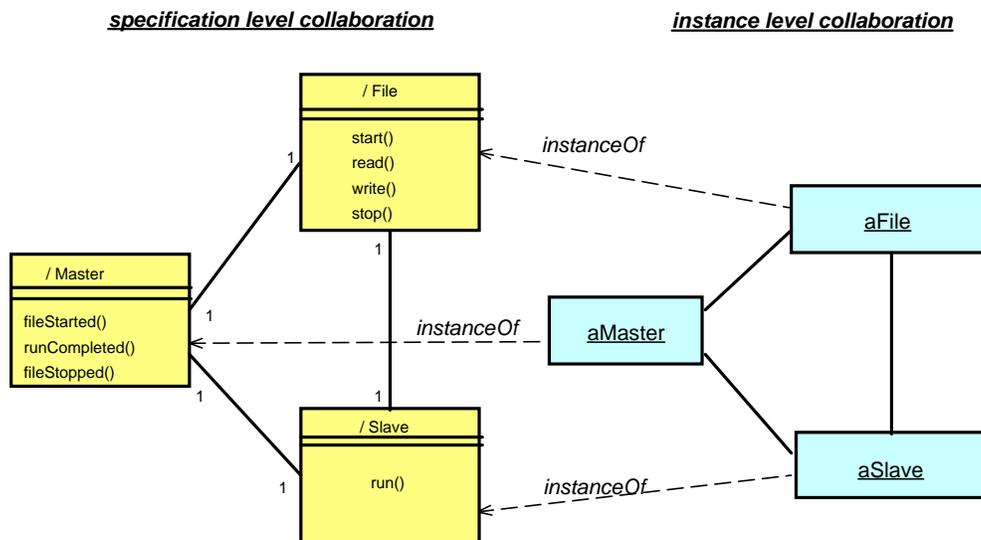


Figure 59. Specification and instance level collaborations example

A message sequence chart for opening the file, editing it, and finally closing it could be as follows:

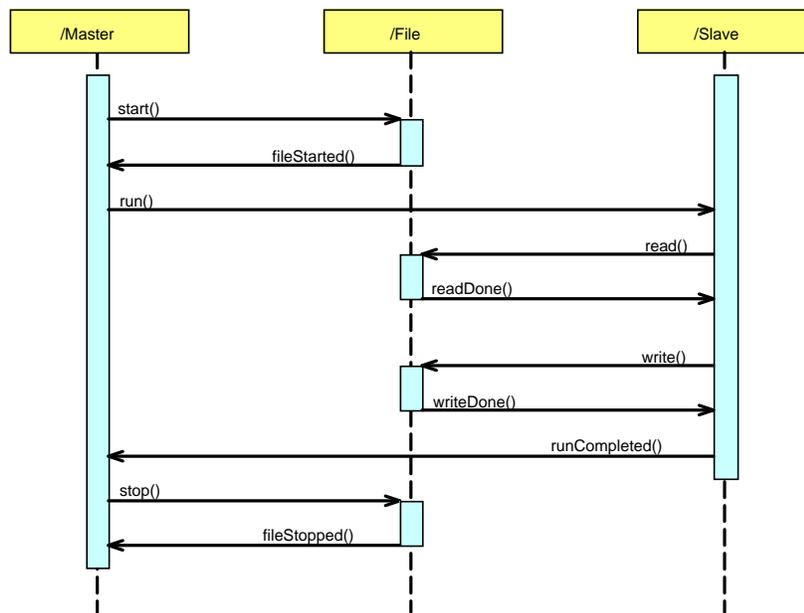


Figure 60. Specification level sequence diagram

This diagram is sufficient for this simple behavior; the file is opened, used, and closed. For more complex behavior, we may need to use state machine diagrams to convince ourselves that the system works properly in all situations. We first create state machines for each ClassifierRole:

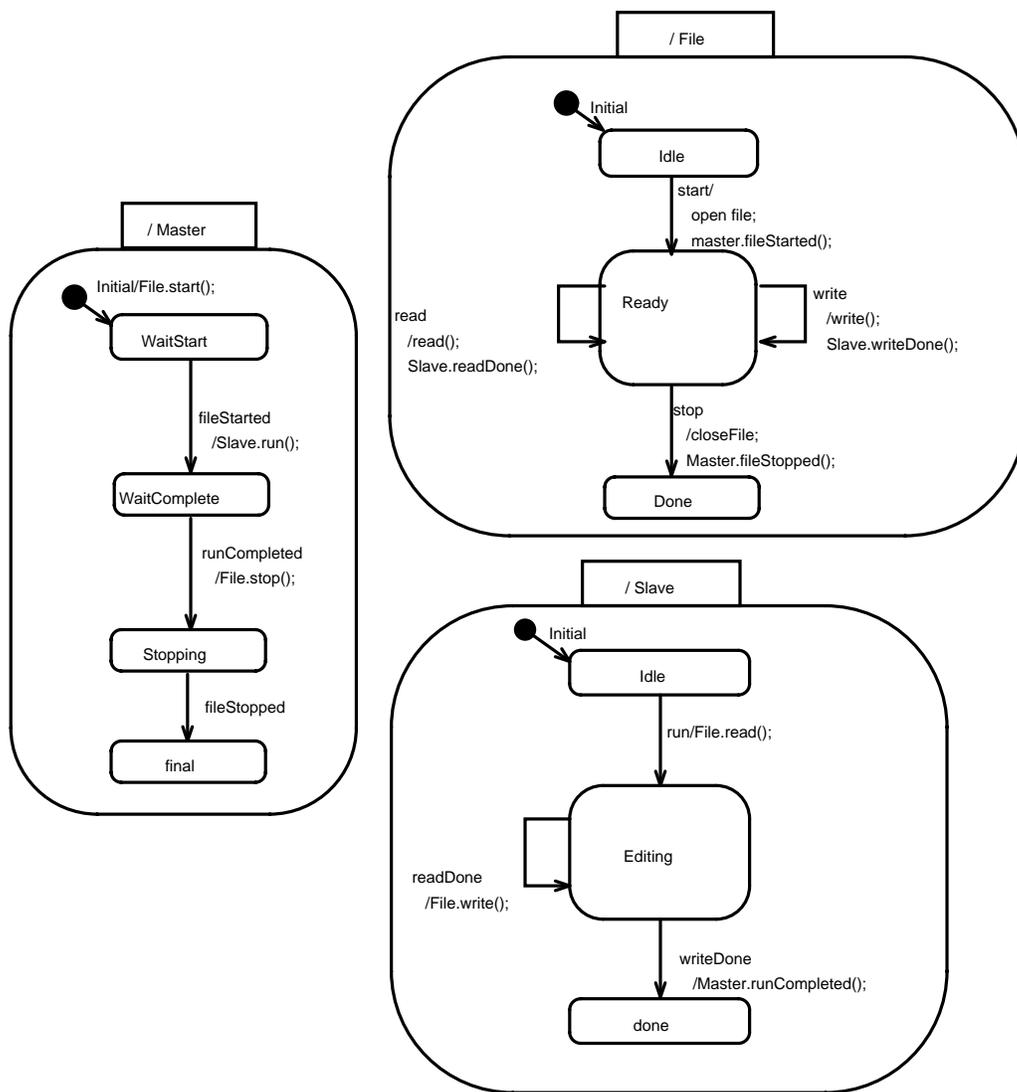


Figure 61. State machines for each of the three ClassifierRoles.

We now reach the crux of the system behavior. We create an overall state machine description of the collaboration as a whole by joining the message sends with the corresponding message receives. We get the following thread of execution:

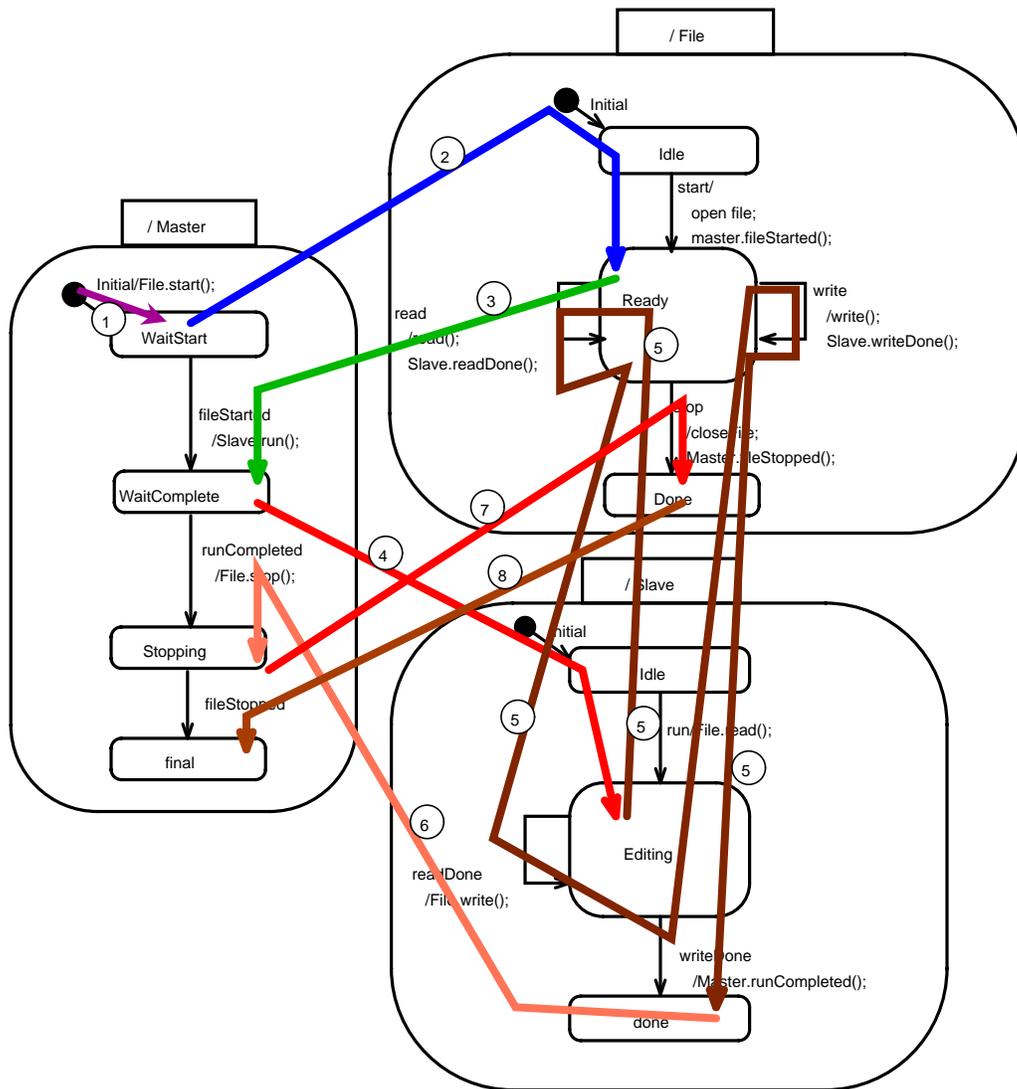


Figure 62. The thread of execution

More formally, we can derive a state diagram for the collaboration as a whole. Each state is defined as the combined states of the three roles; we use the notation *role1-state1* & *role2-state2* & We show every change in the combined state, and the actions associated with the transitions. The result is the following diagram:

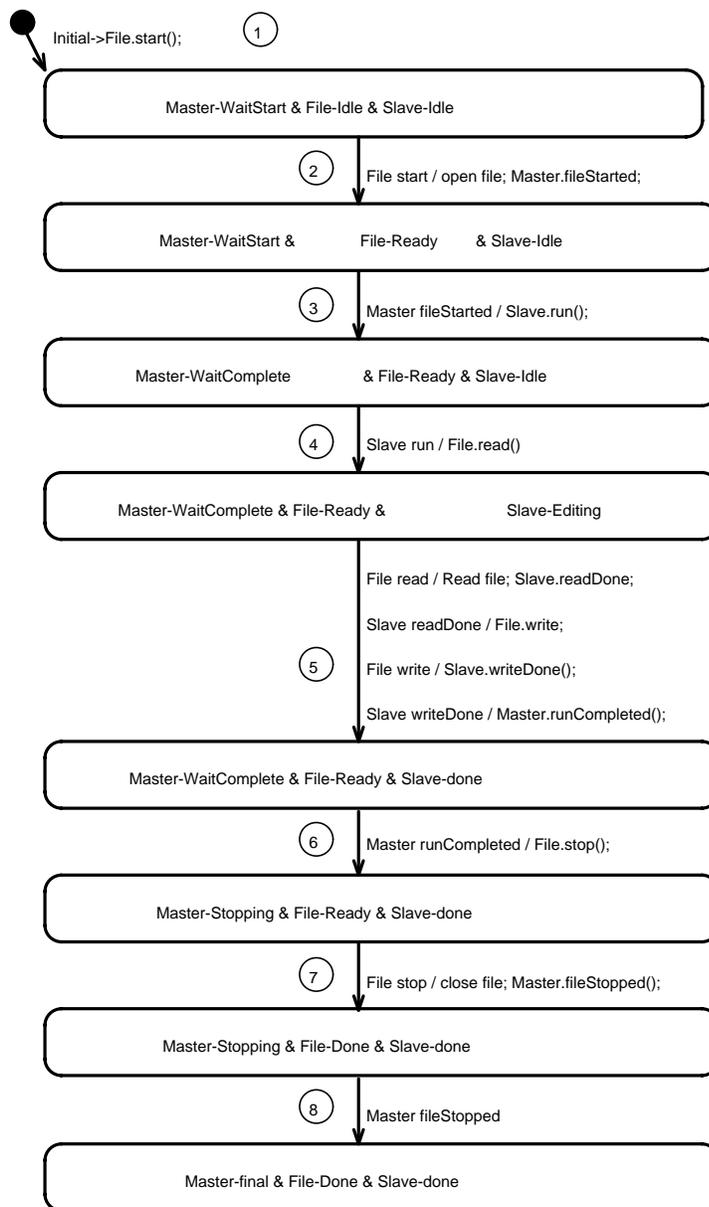


Figure 63. The state machine for the whole system.

This is a trivial example. More complex systems can easily lead to state explosion, making it impractical to draw the above diagram. We are somewhat helped by the separation of concern caused by restricting ourselves to the states that are relevant to the purpose and processes of the a single collaboration. We can get further help by using a special version of the Harel state charts that were proposed by Egil Andersen in his doctoral thesis on role modeling [<ftp://ftp.nr.no/pub/egil/ConceptualModelingOO.ps.gz>]. It can also be argued that we are skating on very thin ice if we cannot oversee the behavior of our system.

In any case, a tool can generate the system behavior from the individual ClassifierRole behaviors and use it to check system consistency. So we now have three levels of consistency checking:

1. The normal typing system can be used by the compiler to check that objects only send messages that will be understood by the receiver.
2. Typed AssociationEnds can be reflected in correspondingly typed references. (The Java language permits typing references with interfaces).

3. A set of state machines that specify the behavior of individual objects can be automatically checked for mutual consistency by a suitable tool.

Up to this point we have only considered very simple cases where there was a one-to-one mapping from role to instance. Now consider that there are any number of slave objects:

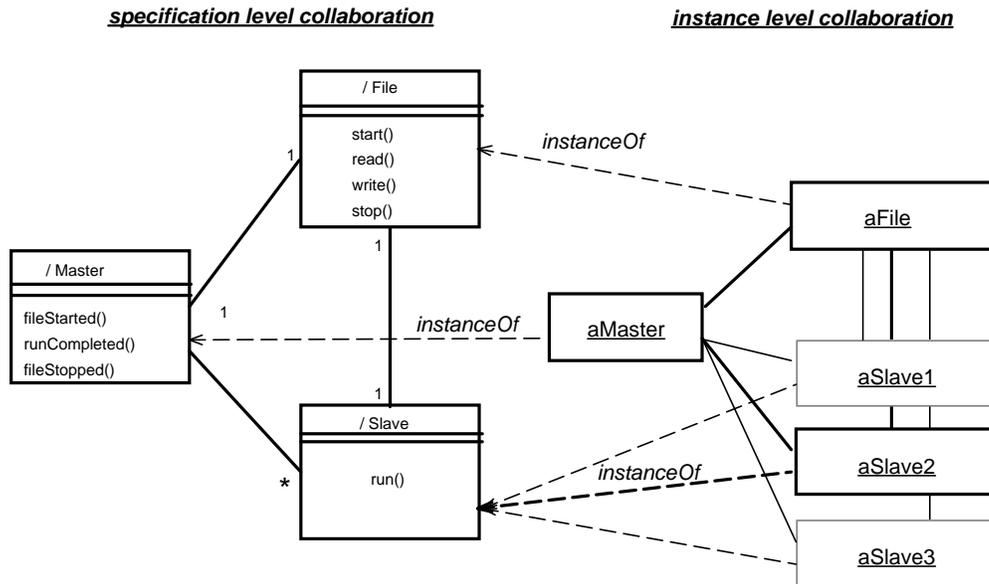


Figure 64. Multiple instances are represented by an archetypical example.

The definition of a collaboration still holds. So one of the slaves is promoted to represent them all. In the above figure, we have designated **aSlave2** to represent the set. The specification means that the other slaves are to behave consistently; that any slave could have been chosen to represent them all; but that once selected, it must represent them throughout the argument. It also means that collaborators waiting for some message from the slave must wait until it has received the message from all the slaves. The complete definition of a Collaboration will then be as follows:

Collaboration [My proposal]

A Collaboration describes how a number of objects work together for a common purpose. There are two aspects. The structural aspect is a description of the responsibilities of each object in the context of the overall purpose of the collaboration; and also the links that connect the objects into a communicating whole. The dynamic aspect is a description of how stimuli flow between the objects to achieve the common purpose.

ClassifierRole [My proposal]

A named slot for an object participating in a specification level Collaboration. Object behavior is represented by its participation in the overall behavior of the Collaboration. Object identity is preserved through this constraint: "In an instance of a collaboration, each ClassifierRole maps onto at most one object."

If there can be more than one instance corresponding to a given ClassifierRole, one of these instances is selected to represent them all. The other instances are constrained to behave in a way corresponding to the selected representative.

The net result is that the above message sequence chart and state diagrams are still valid and need not be changed in any way.

8 Information hiding and component technology

Separation of concerns

We will explore three techniques for separation of concern that are more or less available in UML:

1. *Filtered class diagrams offer unspecified separation.* A class diagram can show a filtered view of an arbitrary selection of classes.
2. *Components separate on services.* A component offers certain services (operations) to its clients while it hides the realizations of these services.
3. *Collaborations separate on system behavior.* One collaboration model describes how a system of interacting components realize one or more operations or use cases.

8.1 Filtered class diagrams offer unspecified separation

For an interesting system, the complete class diagram will be very complex. It is therefore permissible to show arbitrarily filtered views of the diagram. Classes, features, and associations may be hidden in order to highlight some interesting aspect of the system

In UML 1.3 *Semantics*, section 2.3.4, we find:

⌘ *Every time a word coinciding with the name of some construct in UML is used, that construct is referenced.*

So it is quite simple, if somewhat tedious, to find out the full meaning of a term such as *classifier*. Start on the first page and read the complete document carefully. Collate all the places where the term is used, and you have the complete description.

Filtered class diagrams is the weakest technique for separation of concern because the separation is arbitrary and there is no way to convince oneself that they fit together.. When possible, they should be replaced by one or both of the other techniques.

8.2 Separation of concern with components

The concept of components mean different things to different people. The specification of Enterprise Java Beans approaches the ideas from the perspectives of different stakeholders as illustrated in [figure 51](#).

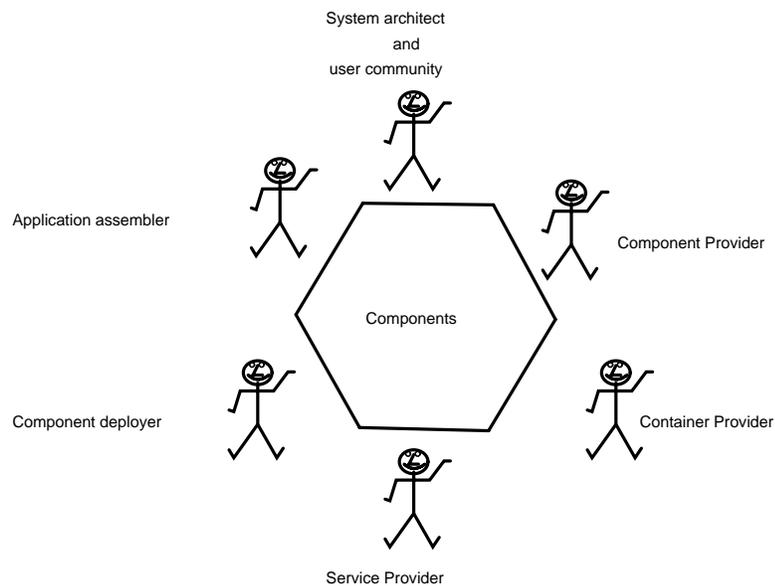


Figure 65. The idea of components mean different things to different people

There are many definitions of components such as Enterprise Java Beans. To my mind, the essence is the the component as seen from the perspective of the system's architect and the system's user community. To them, a component is essentially a "super-object"; an encapsulated realization of a service:

1. A Component has a single access point (an object ID) and offers a well-defined interface to its clients.
2. A Component is reused by cloning
3. A Component does not make assumptions about its clients
4. A Component plays a standardized role within a container
5. Tools are used to deploy components and compose systems
6. Components can contain components. (Follows from the principle of object encapsulation).

Components serve the separation of concern to the extent that they hide complexity and offer well-defined interfaces to their clients.

Figure 52 illustrates two chained components, the second one being itself composed from three components.

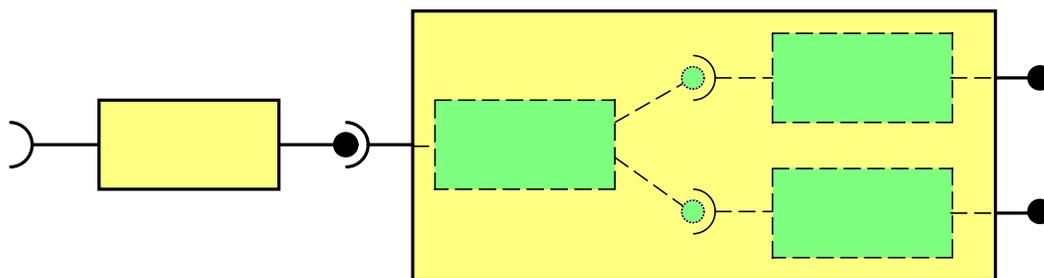


Figure 66. The object nature of components

Standards for shipping objects are still immature. We therefore usually package components in a more primitive form. An example is to use a Java .jar-file containing the necessary classes and other resource files together with rules for their instantiation.

One can think of the component technology as a horizontal separation of concern as illustrated in figure 53.

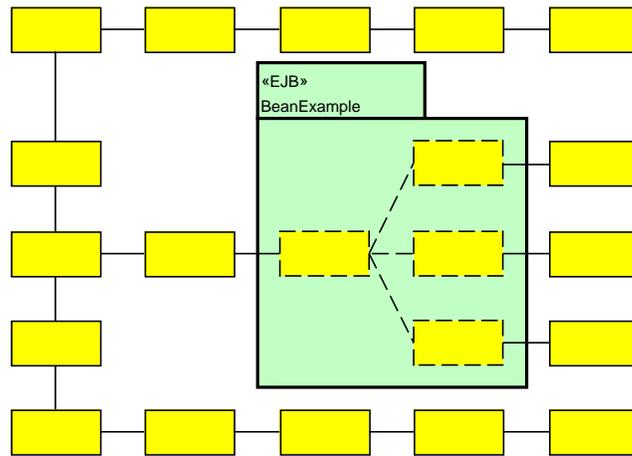


Figure 67. Horizontal separation of concern by object clustering in the horizontal plane.

Work is in progress to find more precise ways of representing components in UML. Two possible starting points are the *UML component* and the *UML subsystem*. For our purposes, the subsystem is the best choice because it is a child of classifier. It can therefore be the base of a ClassifierRole and our semantics is still valid.

8.3 Separation of concern with collaborations

The collaboration models how a society of collaborating objects realizes certain behavior. The separation of concern is achieved by letting each collaboration specify the part of the total behavior and the state that supports this behavior:

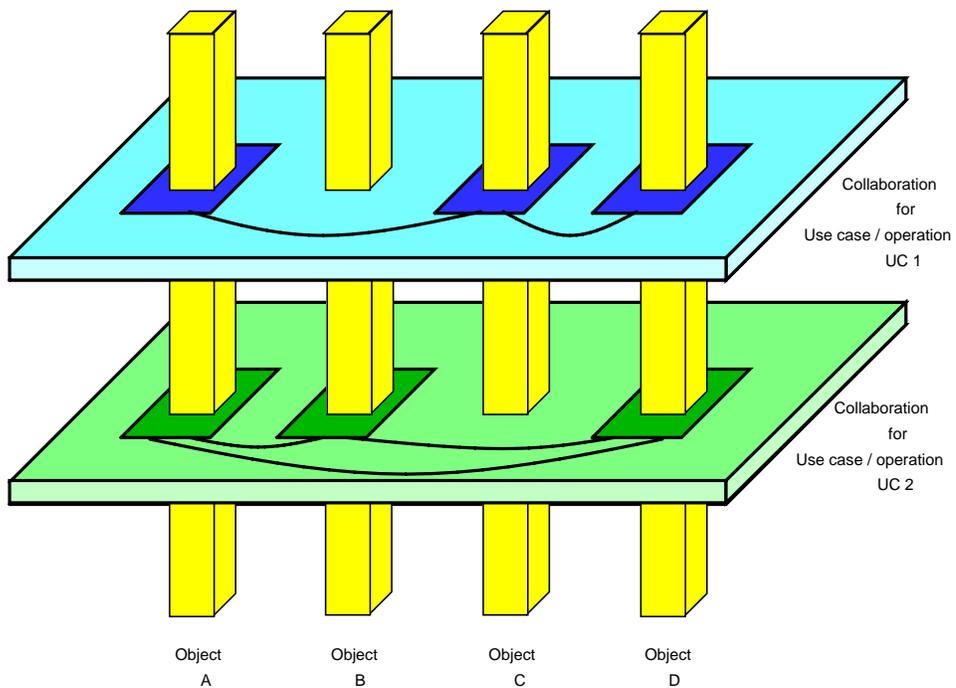


Figure 68. Vertical separation of concern by collaborations

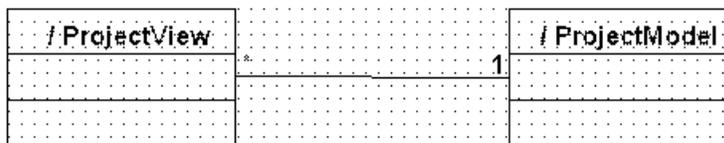


Figure 69. Design example #1 with rich view-model interface

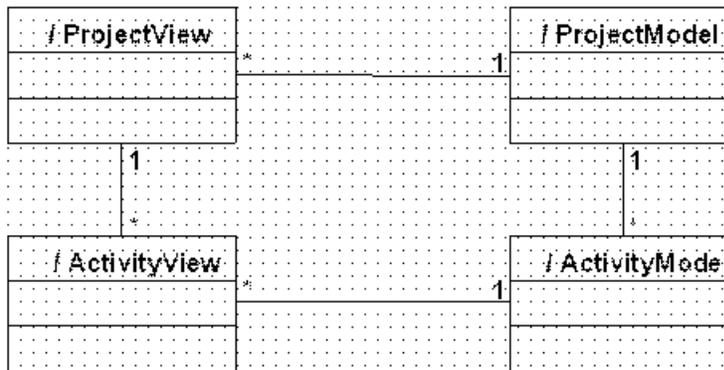


Figure 70. Design example #2 where activity views talk to activity models

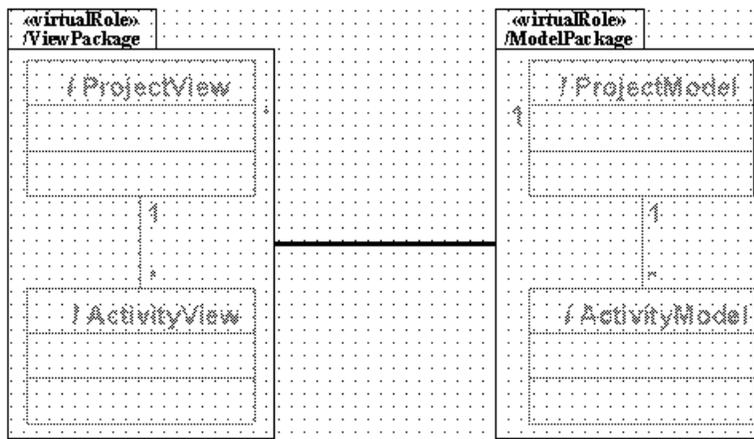


Figure 71. Design example #2 where decision is hidden within components (virtual roles)

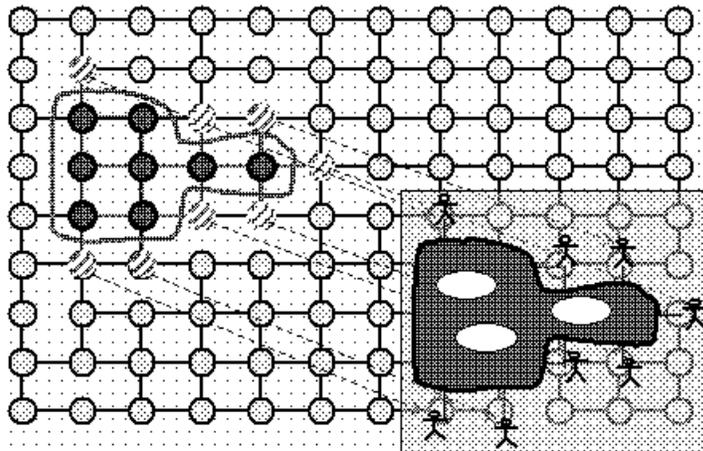


Figure 72. UML Use Case Actors models environment objects

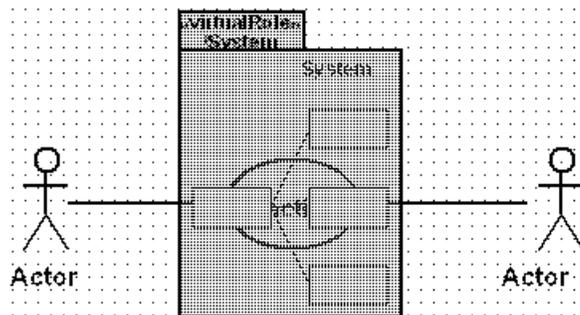


Figure 73. VirtualRoles as a Use Case system

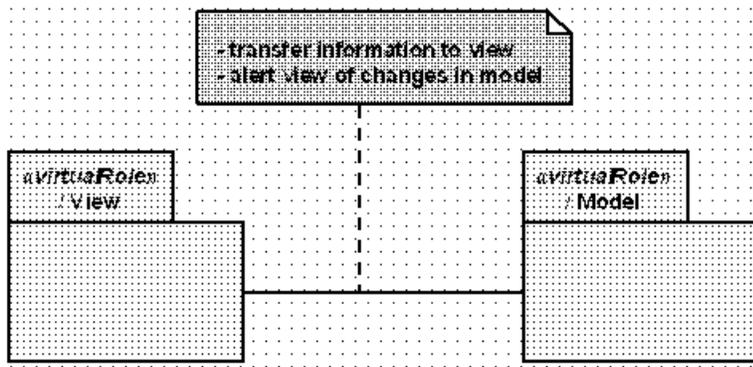


Figure 74. Virtual interactions are not yet standardized in UML

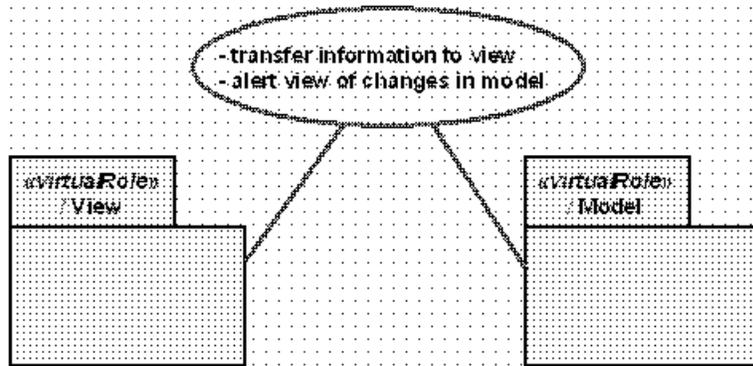


Figure 75. Virtual Interactions a la Catalysis

App. 1 Glossary

1. **action** [UML 1.3 glossary]
The specification of an executable statement that forms an abstraction of a computational procedure. An action typically results in a change in the state of the system, and can be realized by sending a message to an object or modifying a link or a value of an attribute.
2. **association** [UML 1.3 glossary]
The semantic relationship between two or more classifiers that specifies connections among their instances.
3. **class** [UML 1.3 glossary]
A description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment. See: *interface*.
4. **collaboration** [UML 1.3 glossary]
The specification of how an operation or classifier, such as a use case, is realized by a set of classifiers and associations playing specific roles used in a specific way. The collaboration defines an interaction. See: *interaction*.

collaboration [my proposal]
describes how a number of objects work together for a common purpose. There are two aspects. The structural aspect is a description of the responsibilities of each object in the context of the overall purpose of the collaboration; and also the links that connect the objects into a communicating whole. The dynamic aspect is a description of how stimuli flow between the objects to achieve the common purpose.
5. **instance** [UML 1.3 glossary]
An entity to which a set of operations can be applied and which has a state that stores the effects of the operations. See: *object*.
6. **interface** [UML 1.3 glossary]
A named set of operations that characterize the behavior of an element.
7. **message** [UML 1.3 glossary]
A specification of the conveyance of information from one instance to another, with the expectation that activity will ensue. A message may specify the raising of a signal or the call of an operation.
8. **object** [UML 1.3 glossary]
An entity with a well-defined boundary and identity that encapsulates state and behavior. State is represented by attributes and relationships, behavior is represented by operations, methods, and state machines. An object is an instance of a class. See: *class*, *instance*.
9. **operation** [UML 1.3 glossary]
A service that can be requested from an object to effect behavior. An operation has a signature, which may restrict the actual parameters that are possible.
10. **reference** [UML 1.3 glossary]
 1. A denotation of a model element.
 2. A named slot within a classifier that facilitates navigation to other classifiers. Synonym: *pointer*.

11. **role** [UML 1.3 glossary]

The named specific behavior of an entity participating in a particular context. A role may be static (e.g., an association end) or dynamic (e.g., a collaboration role).

ClassifierRole [My proposal]

A named slot for an object participating in a specification level Collaboration. Object behavior is represented by its participation in the overall behavior of the Collaboration. Object identity is preserved through this constraint: "In an instance of a collaboration, each ClassifierRole maps onto at most one object."

If there can be more than one instance corresponding to a given ClassifierRole, one of these instances is selected to represent them all. The other instances are constrained to behave in a way corresponding to the selected representative.

12. **signal** [UML 1.3 glossary]

The specification of an asynchronous stimulus communicated between instances. Signals may have parameters.

1. **The class.** Any abstraction is based on a choice of what is important and what isn't. The highly successful and useful class abstraction is the result of one such choice. It is useful for modeling many interesting aspects of objects. A notable exception is the modeling of system behavior; the class abstraction deals in sets of objects and is not suited for modeling how an object sends a stimulus (a signal or an invocation of an operation) to another object as part of an overall conversation.
2. **The instance level collaboration.** The instance level collaboration shows how objects work together for a common purpose. It models their conversation very nicely, but it is too concrete and specific for all but the simplest cases. I propose an extension of UML 1.3 that defines the abstract syntax of the instance level collaboration and binds it to the existing specification level collaboration in Appendix 1.
3. **The specification level collaboration.** The specification level collaboration is more powerful because it is more abstract. It retains the capability of modeling object conversation and adds capabilities for modeling generalized interaction patterns. It also handles cases with cardinality more than one; this is discussed in [section 8](#).
4. **Separation of concern.** Three techniques are discussed: The arbitrary filtering permitted for class diagrams; components that separate on services, hiding their complexity; and collaborations that separate on system behavior.
5. **Higher level collaboration abstractions with virtualRoles.** Collaborations are often too precise and detailed for overviews and early architecture work. VirtualRoles (stereotype of *package*) permit arbitrary grouping of classifier roles. This hides the detailed object structure without weakening the collaboration semantics.
6. **Abstract interactions.** The basic collaboration specifies interactions in terms of message flow. This may be too detailed for overviews and early architecture work. I propose to use UML comments to name interactions and identify their communication paths (AssociationRoles).
7. **Typing the AssociationEndRoles.** Typing the object references gives improved control over architecture and system design.
8. **The Object-Role-Class trichotomy.** Quite simple, really. (I do not discuss the use of the forward slash):
 "The concrete object"
 / "The role it plays in the collaboration"
 : "The class or classes that can implement it"

9. *System behavior, messages and cardinalities > 1.* An example illustrates how the behavior of the system as a whole can be composed from descriptions of individual ClassifierRole behavior. I also consider the case where there are more than one object playing a given role.
10. *Appendix 1: Instance level and specification level collaborations.* I propose abstract syntax and constraints for the collaboration concept.
11. *Appendix 2: Glossary.* Extracts from the UML 1.3 glossary + a few proposals

1.1 The Object-Role-Class trichotomy

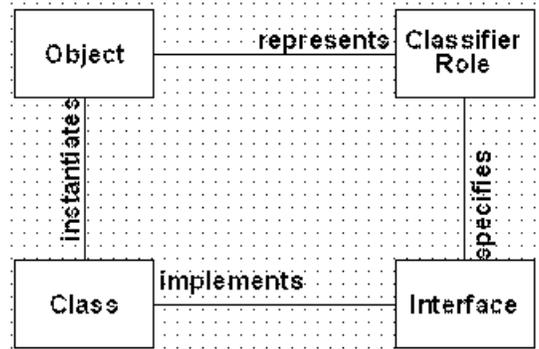


Figure 76.

An object plays a ClassifierRole in a CollaborationSpecification. If we "open up" the ClassifierRole, we will find that it can be realized by one or more classes. As an example, consider the following instances of the *family* Collaboration above:

1. Bjarne /Father : Man & Gina /Mother :Woman & Trygve /Child : Man
2. Trygve /Father : Man & Bjorg /Mother :Woman & Johan /Child : Man
3. Trygve /Father : Man & Bjorg /Mother :Woman & Borghild /Child : Woman

Here are the corresponding Collaboration diagrams. First the "pure" CollaborationSpecification:

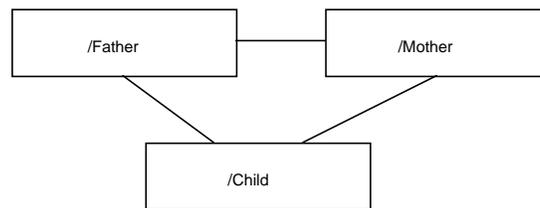


Figure 77. A "pure" specification level collaboration

We next add information about Role realization. *Note that this does not add essential information to the Collaboration as such assuming the Collaboration included interface information.*

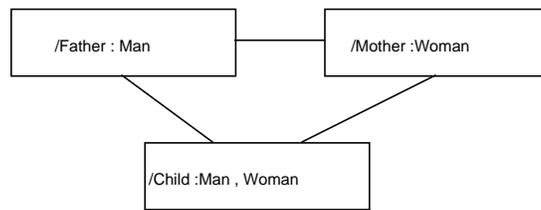
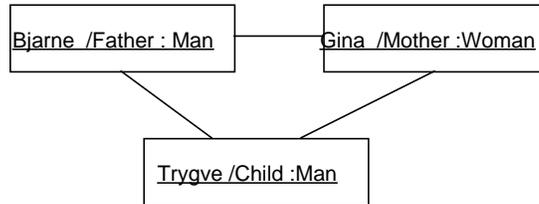


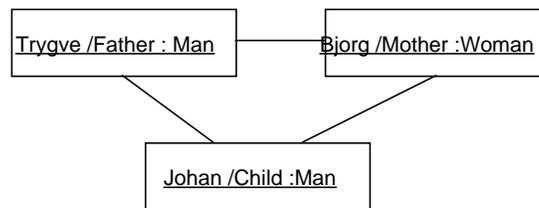
Figure 78. Include specification of class or alternative classes

So Father is of class *Man*, Mother of class *Woman*, and Child of either class.

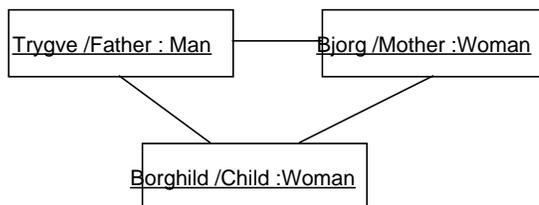
We finally look at a few instances of this Collaboration. We have, somewhat arbitrarily, decided to include the role and class names:



A Collaboration at instance level



Another Collaboration at instance level



A third Collaboration at instance level

Figure 79. Instance level collaboration with specification of ClassifierRole and class

The object/role/class trichotomy is easy to explain. We can use a theater metaphor to explain the difference between role (ClassifierRole) and actor (object) and his capabilities (class). Or, usually even better, we can use a business organization metaphor: John can play the role of *SystemArchitect* in a certain project and belong to the class of *Programmers*. He may, of course, also play other roles in this or other projects.

Instantiating class *Man* gives an instance of *Man*.

In contrast, it is meaningless to instantiate a ClassifierRole in isolation. The role gets its meaning from its position in the Collaboration and can only be instantiated in this context. For example, instantiating the */Father* role involves the following:

1. Select a suitable factory object.
2. Ask the factory object for an instance that has the features needed to play the role.
3. Connect the resulting object into the collaboration structure. For example, this may involve linking the father object to the child and mother objects, as well as linking the mother and child objects to the father object.

App. 2 Comments for programmers

<East Coast approach = Extending progr. language with a smart record.>

<West Coast approach = A new way of thinking about programs and the world in general.>

<Programmer Challenge: transition from east to west. Overall structure dominates, coding becomes a low level mundane task. (Compare transition coder --> database designer. Some people still haven't managed that one)>

<Notice that the classes are invisible on this level. We will understand the objects before considering the coding in terms of classes. Also note that class association diagrams are not satisfactory for our purposes because they link classes and therefore lose object identity. At the top level, we focus on object responsibility and interaction and do not care about object implementation.">

App. 3 References

<http://www.ifi.uio.no/~trygver>

where I work with an example development

also see "documents", particularly "UML Collaboration and OOram semantics"

Reenskaug, Wold, Lehne: *Working With Objects*. Manning/Prentice Hall 1996. ISBN 0-13-452930-8

The reference work (out of print)

Egil P. Andersen: *Conceptual Modeling of Objects. A Role Modeling Approach*. Dr Scient thesis. Dept. of Informatics, University of Oslo. 4 November 1997.

The theory of role modeling

<ftp://ftp.nr.no/pub/egil/ConceptualModelingOO.ps.gz>

Trygve Reenskaug : *Working with objects: A three-model architecture for the analysis of information systems*. JOOP May 1997.

- [Aronson] Elliot Aronson: *The Social Animal*. Freeman, San Francisco 1972. ISBN 0-7167-0829-9.
- [Bjerde, Berre, Oldevik] Lasse Bjerde, Arne-Jørgen Berre, Jon Oldevik: *Describing a system from multiple viewpoints - using ISO/RM-ODP with OOram role modelling and UML*. Workshop position paper, OOPSLA '98
- [E.P.Andersen] Egil P. Andersen: *Conceptual Modeling of Objects. A Role Modeling Approach*. Dr Scient theses. Dept. of Informatics, University of Oslo. 4 November 1997.
Download from <ftp://ftp.nr.no/pub/egil/ConceptualModelingOO.ps.gz>
- [Etzioni 64] Amitai Etzioni: *Modern Organizations*. Prentice-Hall 1964. pp 53-54
- [Gang of Four] Gamma, Helm, Johnson, Vlissides: "Design Patterns". Addison-Wesley 1994. ISBN 0-201-63361-2
- [Goldberg] Goldberg and Rubin: "Succeeding with objects". Addison-Wesley 1995. ISBN 0-201-62878-3
- [Hol 77] Erik Holbæk-Hanssen, Petter Håndlykken, Kristen Nygaard: *System Description and the Delta Language*. Norwegian Computing Center publication no. 523. Second printing, Oslo 1977.
- [IDL] <OMG IDL refs.>
- [Jacobson] Ivar Jacobson: *Object-Oriented Software Engineering*. Addison-Wesley, New York, 1992 (ISBN 0-201-54435-0)
- [Java glossary] <http://java.sun.com/docs/glossary.html> contains the "official" Java glossary. Several of our definitions are copied from this reference.
- [Java Specification] With the publication of The Java Language Specification, James Gosling, Bill Joy, and Guy Steele provide the definitive technical reference for the Java programming language. It provides complete, accurate, and detailed coverage of the entire language and its syntax. If you want to know the precise meaning of Java's constructs, this is the source. Download from <http://java.sun.com/docs/books/jls/>
- [Java Tutorial] A good introduction to the Java language. <http://java.sun.com/docs/books/tutorial/>
- [Netscape/Java] with a working Java platform can be downloaded from <http://developer.netscape.com/software/jdk/download.html>
- [Norman] Donald A. Norman: "The Design of Everyday Things". Doubleday. New York 1989. ISBN 0-385-26774-6.
- [Novasoft] http://www.novasoft.com/web/developer/alphabet_soup/index.htm includes a white paper for many key concepts in distributed computing including an extensive glossary.

- [ODP] ISO/IEC, "ISO/IEC 10746-1 Information technology - Basic reference model of Open Distributed Processing - Part 1: Overview," ISO ITU-T X.901 - ISO/IEC DIS 10746-1, 1996.
 [2]ISO/IEC, "ISO/IEC 10746-2 Information technology - Open Distributed Processing - Reference Model:Foundations," , 1996.
 [3]ISO/IEC, "ISO/IEC 10746-3 Information technology - Open Distributed Processing - Reference Model: Architecture," , 1996.
 [4]ISO/IEC, "ISO/IEC DIS 10746-4 Information technology - Open Distributed Processing - Part 3: Architectural semantics," , 1996.
 Retrieve from http://www.dstc.edu.au/AU/research_news/odp/ref_model/ref_model.html
- [OMG 95] Richard M. Soley (ed.), Christopher M. Stone: Object Management Architecture Guide. Revision 3.0. John Wiley & Sons, New York, 1995. ISBN 0-471-14193-3.
 Also see <http://www.omg.org/>.
- [Reenskaug 77] Trygve Reenskaug: *Prokon/Plan -- A ModelingTool for Project Planning and Control*. IFIP Conference, North Holland, New York 1977.
- [Reenskaug 80] Trygve M. H. Reenskaug: User-Oriented Descriptions of Smalltalk Systems. Byte, August 1981 (The Smalltalk issue). pp. 148-166
- [Reenskaug 97] Trygve Reenskaug: *Working With Objects: A three-model architecture for the analysis of information systems*. JOOP, May 1997. Retrieve from <http://www.ifl.uio.no/~trygve/documents/95Article/951010-paper.ps>
- [ReeWoLeh] Reenskaug, Wold, Lehne: *Working With Objects*. Manning/Prentice Hall 1996. ISBN 0-13-452930-8
- [RMI] Java Remote Method Invocation. Tutorial, specification and more at <http://java.sun.com/products/jdk/1.1/docs/guide/rmi/index.html>
- [TINA] Trygve Reenskaug: *The Industrial Creation of Intelligent Network Services*. TINA '93 - The Fourth Telecommunications Information Network Architecture Workshop, L'Aquila, Italy, September 1993. Proceedings from Scuola Superiore G. Reiss Romoli S.p.A.: Str. Prov. per Coppito km 0,300; 67010 Coppito (AQ); Italy.
- [UML] Unified Modeling Language (UML). Proposal to the Object Management Group. Version 1.3, June 1999. Documents can be retrieved from the OMG at <http://www.omg.org/cgi-bin/doc?ad/99-06-08>
- [XML] **Overview of XML standards** with some relevant links:
http://www2.software.ibm.com/developer/standards.nsf/xml_describing_byname
http://www.w3.org/TR/NOTE_dcd (W3C Document Content Description for XML)
<http://www.geocities.com/WallStreet/Floor/5815/guide.htm> (Guidelines for using XML for Electronic Data Interchange)
<http://www.geocities.com/WallStreet/Floor/5815/Book.htm>
<http://msdn.microsoft.com/xml/tutorial/default.asp> (MS XML tutorial)
Other resources:
<http://www.alphaworks.ibm.com/>
Articles
<http://www.devx.com/upload/free/features/webbuilder/1999/wb0599/kc0599/10min0599ie5.htm>
 (create intuitive tables in IE5 using XML, data islands, and behaviors)

TABLE OF CONTENTS

1	Introduction	1
2	What it is all about	2
2.1	The dream of the bug free computer system.....	5
2.2	The dream of putting the user in the driver's seat.....	8
2.3	The OMG dream of a world of objects.....	13
2.3.1	The object structure as a rational organization.....	14
2.3.2	Modeling systems of collaborating objects.....	16
3	Modeling with objects	22
4	The specification level Collaboration	34
5	Typing the AssociationEndRoles	42
6	Specialization of a Collaboration	45
6.1	MVC example. note dated 17 June 1997.....	48
6.1.1	Model diagrams of MVC and TextEditor.....	49
6.1.2	Metamodel considerations.....	52
7	System behavior, messages and interfaces	55
8	Information hiding and component technology	61
	Separation of concerns	
8.1	Filtered class diagrams offer unspecified separation.....	61
8.2	Separation of concern with components.....	61
8.3	Separation of concern with collaborations.....	63
App. 1	Glossary	67
1.1	The Object-Role-Class trichotomy.....	69
App. 2	Comments for programmers	72

INDEX

figure 5.....	4
figure 37.....	38
action.....	67
association.....	67
class.....	22, 67 , 69, 70
ClassifierRole.....	36, 60, 68
Collaboration.....	26, 36, 60, 67
exampleActivityNetwork.html.....	4
figure 1.....	2
figure 12.....	12
Figure 13.....	14, 24, 25, 33, 34
Figure 14.....	16, 17
figure 15.....	17
Figure 16.....	18, 19
Figure 2.....	3
figure 20.....	22
figure 21.....	23, 26, 42
Figure 22.....	24
Figure 23.....	24, 25, 34
figure 24.....	25
figure 25.....	25, 26
figure 26.....	26, 34
Figure 27.....	27
figure 28.....	29, 43
figure 29.....	30, 36, 37, 39
figure 3.....	3, 8, 17
figure 31.....	31
figure 32.....	34
figure 33.....	35, 36
figure 34.....	35
figure 35.....	36, 37, 39, 42
figure 36.....	37, 42
figure 37.....	37
figure 38.....	39
figure 39.....	39

figure 4	3, 8, 27
figure 40	39, 41
figure 5	19, 29
figure 51	61
Figure 52	62
figure 53	63
Figure 6	7
Figure 7	8, 11
figure 8	9, 11
Figure 9	10
ftp://ftp.nr.no/pub/egil/ConceptualModelingOO.ps.gz	59, 73
http://developer.netscape.com/software/jdk/download.html	73
http://java.sun.com/docs/books/jls/	73
http://java.sun.com/docs/books/tutorial/	73
http://java.sun.com/docs/glossary.html	73
http://java.sun.com/products/jdk/1.1/docs/guide/rmi/index.html	74
http://msdn.microsoft.com/xml/tutorial/default.asp	74
http://www.dstc.edu.au/AU/research_news/odp/ref_model/ref_model.html	74
http://www.geocities.com/WallStreet/Floor/5815/Book.htm	74
http://www.geocities.com/WallStreet/Floor/5815/guide.htm	74
http://www.novasoft.com/web/developer/alphabet_soup/index.htm	73
http://www.omg.org/	74
http://www.omg.org/cgi-bin/doc?ad/99-06-08	74
http://www.w3.org/TR/NOTE dcd	74
http://www2.software.ibm.com/developer/standards.nsf/xml describing byname	74
instance	67
interface	42, 67
message	67
Object	14, 67 , 69
operation	67
planningExample-0.zip	4
reference	67
role	68
section 2.3.2	27
section 5	33
section 8	68
signal	68

[Aronson].....	73
[Bjerde, Berre, Oldevik].....	73
[Dijkstra 1976].....	5, 6
[E.P.Andersen].....	73
[Etzioni 64].....	16, 73
[Gang of Four].....	73
[Hol 77].....	18, 73
[IDL].....	17
[IDL].....	73
[Java glossary].....	73
[Java Specification].....	73
[Java Tutorial].....	73
[Netscape/Java].....	73
[Norman].....	8, 73
[Novasoft].....	73
[ODP].....	74
[OMG 95].....	13, 74
[Reenskaug 77].....	3, 74
[Reenskaug 80].....	74
[Reenskaug 97].....	74
[RMI].....	74
[UML].....	74
[XML].....	18, 74