Augment static classes by dynamic roles in the BabyUML discipline of programming
Trygve Reenskaug
Dept. of Informatics, University of Oslo

Abstract
The goal of the BabyUML project is to increase my confidence in my programs. The keywords are leverage and simplicity. Leverage lets me say more with less. Simplicity helps me think clearly and a reader understand and audit my code; my peers are very good at catching my blunders.

The essence of object orientation is that objects interact to produce some desired result. Yet current programming languages are focused on individual object as they are specified by their classes; there are no explicit language constructs for describing communities of interacting. In BabyUML, we zoom back from the individual objects and let our code specify the community as a whole with abstractions taken from OOram role modeling or the UML concepts of collaboration and interaction.

The BabyUML project is experimental; its ideas and concepts are explored and their feasibility demonstrated with actual code and running programs. This article describes two BabyUML experiments.

One experiment explores an old and a new paradigm for organizing objects in clear and explicit structures that I illustrate with a running Java program. The old paradigm is the MVC, the Model-View-Controller. MVC describes the roles played by objects that bridge the gap between a human mental model and the corresponding data stored in the computer. The new paradigm is the DCA, the Data-Collaboration-Algorithm. DCA differentiates model objects according to clearly defined responsibilities and makes object structures and interactions explicit in the code.

A class describes the nature of an object. A role describes an object as it is seen in an ensemble of interacting objects. The class is static, the nature of an object stays the same throughout the object’s lifetime. The role is dynamic, a role may be played by different objects at different times. The static/dynamic dichotomy is the foundation of BabyUML. The goal is a new discipline of programming in which the two are balanced so as to give the programmer new capabilities and new leverage.

The first steps of another experiment is discussed in the last part of the chapter. The experiment shall lead to a new discipline of programming where programmers can work consistently at a high conceptual level throughout coding, debugging, testing and maintenance. The experiment shall include high-level languages, metaprogramming, and IDEs; all implemented in a corner of Smalltalk that I have called the BabyUML Laboratory. Smalltalk is chosen because its classes and metaclasses are programmed as regular objects. We describe the laboratory and how it can support metaprogram-
ming by specializing metaclasses and tools to provide a consistent, high-level BabyUML discipline of programming. We finally indicate the future direction towards a workable BabyIDE.

Appendix (on CD): The Java experiment code.
1 INTRODUCTION

On the 9th September 1945, a moth was found trapped between the contact points on relay #70, Panel F, of the Mark II Aiken relay calculator. The event was entered in the calculator’s log book as the word’s first record computer bug [Hopper-45]. This first bug was an “act of God”; most later bugs are blunders of our own making and the fight against them has been an essential part of software engineering ever since. Consider the following quotes from the first NATO Software Engineering conference [NATO-68]:

**David and Fraser:** Particularly alarming is the seemingly unavoidable fallibility of large software, since a malfunction in an advanced hardware-software system can be a matter of life and death.

**Dijkstra:** The dissemination of knowledge is of obvious value -- the massive dissemination of error-loaded software is frightening.

The same could have been uttered today. The needs of society are still beyond us. The insatiable software market is ever wanting more, and we keep promising more than we can deliver. Many years ago, the Oslo public transport company announced a new ticketing system. As a user of their services, I found the promised functionality very advanced and very attractive. New card readers were installed at all stations. A date for transition to the new system was announced. But software was delayed for a year, then another. Today, the card readers stand idle, the next deployment date is in 2008.

In his 1980 Turing Award lecture, Tony Hoare succinctly stated our choices [Hoare-81]:

“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 is to make it so complicated that there are no obvious deficiencies.”

The second way is the easy way. We get it by default when we fail to find a simple design. May be the time available is unreasonably short. Or may be our concepts, languages, and tools do not match the challenges posed by the requirements. We end up relying on testing to get most of our blunders out of the system. But any given test method can only find a certain percentage of the all errors. So keeping the test method fixed, the more errors we find during testing, the more errors are probably left in the shipped software. To quote Dijkstra:

"Program testing can be used to show the presence of bugs, but never to show their absence!" [Dijkstra-structured]

and

“One of the reasons why the expression "software industry" can be so misleading is that a major analogy with manufacturing fails to hold: in software, it is often the poor quality of the "product" that make it so expensive to make! In programming, nothing is cheaper than not introducing the bugs in the first place." [Dijkstra-next50]

Hoare’s first way is the hard way. It is also the only way to get quality software, because no industry has ever been able to work quality into an inferior product by testing it. I have been programming for half a century and simplicity has always been my holy grail. The simple structure is not only the key
to mastery by my brain, but also the key to a correspondence between user requirements and system implementation and thus to habitable systems.

Through the years, requirements have escalated from the simple computation to distributed systems with complex data and powerful algorithms. My brain has remained roughly the same, so I have had to rely on better tools for thinking, designing, coding, maintaining, etc. My tools have been ahead of the requirements some of the time, and I have had the deep satisfaction of running tests merely to check that I haven’t made serious blunders. At other times, requirements have been ahead of my tools, and I have shamefully been forced to rely on testing to get some semblance of quality into my programs.

Requirements have been ahead of my capabilities for quite some time now. I started the BabyUML project in an attempt to remedy this deplorable situation, hoping once again to experience the deep satisfaction of following Hoare’s first way. I envision a high level programming discipline that facilitates compact and simple descriptions of significant systems of collaborating objects. I want to be able to write a piece of code and give it to a colleague so that she can audit it and take responsibility for its correctness. I want my code to be effectively chunked and self documenting so that other people can read it and grasp its architecture and operation. Lines of code and even efficiency come second. I claim that while low level code can be very efficient in the small; high level code is superior in the large because it is more readable and thus makes it easier to find efficient structures. Further, high level code is also easier to change to reflect new insights in design or requirements.

My brain is roughly the same as it has always been. My challenges have grown, and I need better support that can take the form of an improved metamodel that extends the kinds of programs my brain can cope with. The program descriptions stored in the computer will be instances of the same metamodel and I will require improved tools to bridge the gap between my brain and the program description. The BabyUML success criterion is that programmers shall be happier and more effective when they use its results. Programmer happiness is closely coupled with responsive environments; exploration and evolution is clearly better than being forced to live with old decisions.

The Baby was the world’s first electronic, digital, stored program computer. It executed its first statements on the 21st June 1947 at the University of Manchester, England. BabyUML is, somewhat whimsically, named after this computer because is based on the idea of a stored program object computer such as it is pioneered in Smalltalk. The other part of the name, UML, reflects that I see UML as a gold mine of concepts and ideas that are unified into a fairly consistent metamodel, many of them applicable to the project.

The BabyUML project is experimental because I find that trial and error is a better toolmaker strategy than theory followed by implementation. The result of the BabyUML experiments shall be a new discipline of programming that includes tools, metamodels, and processes. One or more new programming languages may or may not be required.

The first series of experiments is based on a simple example taken from planning and control with activity networks as described in section2A. In
section 2B, I use the example to illustrate why my old programming style fails to scale up to my current problems.

I expect to find most of the concepts I will be needing in UML and in many other places, so I will select and adapt rather than invent. In section 3, I have selected some fundamental ideas that have proven their worth in the past and discuss them from a BabyUML perspective.

I cannot device a new discipline of programming before I understand what I want to achieve, i.e. the run time structure of interacting objects. BabyUML will provide leverage by restricting the programming environment to support an extensible set of object structuring paradigms. The main part of this chapter is the discussion of a Java implementation of my old MVC and my new DCA, two paradigms that exemplify the kinds of object structures I envisage for BabyUML. Both paradigms answer the critical questions: What are the objects, How are they interlinked, and How do they interact. Both are important stepping stones in my pursuit of the utmost simplicity.

MVC is an old paradigm that has survived for more than 30 years. The MVC bridges the gap between the human brain and the domain data stored in the computer. Its fundamental quality is that it separates model from view, i.e., tool from substance. The ideal Model is pure representation of information, while the ideal View is pure presentation:

- The domain data are represented in an object called the MVC Model.

- The human user observes and manipulates the data through an MVC View. The view shall ideally match the human mental model, giving the user the illusion that what’s in his mind is faithfully represented in the computer.
- The MVC Controller is responsible for setting up and coordinating a number of related views. The Controller is sometimes called a Tool.

The Java example implementation of the MVC is discussed in section 4.

The Java MVC and DCA implementations are both subordinated the overriding idea of partitioning the object space into components. The BabyUML component is a “monster object” that looks like a regular object in its environment where it is completely characterized by its provided interface. Inside, the component implementation can obey different paradigms; here exemplified by MVC and DCA. The notion of a BabyUML component is recursive; its encapsulated objects can turn out to be components in their own right without this being apparent from their external properties. The partitioning of the total system into components is an important contribution to simplicity.

An example implementation of the DCA paradigm is discussed in section 5. Objects that are encapsulated by a component may be organized in many ways. The DCA paradigm prescribes a particular way based on object responsibilities. The D for Data part is a set of objects that instantiate a conceptual schema for the domain objects. The A for Algorithm part is a set of methods that specify domain object interactions for all provided operations. The C for Collaboration part is a set of

1. The program is given in full on the enclosed CD.
classes specifying external data schemas that provide access methods tailored to simplify data access in the Algorithms.

The DCA paradigm illustrates a dicothomy between classes that specify the properties of objects and roles that specify the contributions made by objects when they interact to reach a common goal. The class is thus a static property of an object, while the role is dynamic. A class describes the inside of an object. A role describes an object as it is seen in an ensemble of interacting objects. The class/role dichotomy is the foundation of BabyUML. The goal is a new discipline of programming where the two are balanced so as to give the programmer new capabilities and new leverage.

The experiments reported in section 4 and section 5 are completed. They have revealed the kind of high-level structures that shall be included in the BabyUML discipline of programming. The next major step is to experiment with the BabyIDE -- an integrated development environment for BabyUML. In the experiment, I will try out novel semantics for classes and metaclasses together with tools for design, compilation, and inspection. The first part of this step is to establish a BabyIDE laboratory embedded within the Smalltalk environment. Its main features is that it gives the systems programmer full control over the semantics of classes and metaclasses. It transpired that it was essential with a deep understanding of the of the notions and implementation of objects, classes, instantiation and inheritance. The implementation of a rudimentary laboratory has been completed, and the resulting understanding is reported in section 6.

The next major step will be to create a BabyIDE for the DCA and MVC paradigms. I will clearly need to harness imperative, algorithmic programming as well as the declarative definition of data structures. I will need class oriented programming to define the nature of the objects as well as role models (collaborations) to define their interaction. I will also need new debuggers and inspectors to create an integrated environment. The prospects are challenging, and I look forward to dig into them. Section 6 describes the core concepts of the BabyIDE implementation.
2 AN EXAMPLE, A PROBLEM, AND A SOLUTION

2A An Activity Network Planning Example

Project planning and control is frequently based on the idea of activity networks. A piece of work that needs to be done is described as an activity. The work done by an architect when designing a house can be broken down into activities. The work of erecting the house likewise. Example activities: drawing a plan view, digging the pit, making the foundation, erecting the frame, paneling the walls, painting these walls.

An activity is characterized by its name, its duration, its earlyStart and earlyFinish times, its lateStart and lateFinish times, a set of predecessor activities, and a set of successor activities. Predecessors and successors are called technological dependencies. An activity can start when all its predecessors are finished, and a successors cannot start before the current activity is finished. There are more sophisticated forms of technological dependencies. For example, it is possible to start the painting of one wall before the panelling of all walls is finished. Such cases are catered for with various kinds of activity overlap.

A start activity is an activity with no predecessors. Frontloading is the calculation of the early start and finish times of each activity given the start time of the start activities. Similarly, an end activity is an activity with no successors. Backloading is the calculation of the late start and finish times of each activity given the end time for the end activities.

Activities may also be tied to resources. The creation of a design drawing requires some hours of work by an architect and a draftsman. The digging of the pit requires machinery and the efforts of some navvies. Resource allocation is to reserve resources for each activity. A scarce resource may delay the whole project. Resource allocation is a non-trivial operation; one can easily end up with unimportant activities blocking the progress of critical ones. (We cannot dig the pit because the navvies are busy levelling the garden.)

The example chosen for this experiment is the rudimentary activity network shown in figure 1. The activity duration, earlyStart and earlyFinish times are shown in parenthesis. There is a single resource; say a pool of workers. It has unlimited capacity and an activity employs a single worker for its duration.

![Fig. 1: The experimental activity network.](image)

The Java program GUI is shown in figure 2. It’s partitioned into four strips. The top strip has three command buttons: Create First network (the one shown in figure 1). Frontload the network and allocate resources. Create Second network is for demonstrating that the program works for more than one network. The second strip shows the dependency graph. The third strip is a gantt diagram showing when the different activities will be performed. Time along the horizontal axis, activities
along the vertical. The bottom strip shows how the activities are allocated to the resource. Time along the horizontal axis, resource loading along the vertical.

![Diagram of Java program user interface]

The above example could be programmed in many different ways. I will later use it to illustrate the MVC and DCA paradigms, pretending that I’m working on a non-trivial, comprehensive planning system.

### 2B My old style doesn’t scale

A potential problem with my usual programming style is easily demonstrated. Figure 3 illustrates how I normally implement the network example. The rounded rectangles denote objects, the solid lines denote links between them, the white rectangles denote classes, and the dotted arrow denotes «instanceOf».

The activity objects are shown bottom right with heavy outlines. The idea is that planning is realized by negotiation; internally between the activity objects themselves and externally between activity objects and their required resources (manpower, machinery, etc.) The technicalities of the user interface have been separated from the domain objects; the GUI objects are shown on the left.

My implementation style tends to give fairly small objects in a distributed structure and with distributed control. It leads to a large number of links and complex interaction patterns. An activity uses a certain resource; let the activity object negotiate directly with the resource object to establish a mutually acceptable schedule. A symbol on the computer screen represents a certain activity; let the symbol object interrogate the activity object to determine how it is to be presented, and let the activity object warn the symbol object of significant changes. Large systems can degenerate into a bowl of spaghetti.

![Diagram of a typical application]

Every object is an instance of a class written in a language such as Simula, Java, or Smalltalk. The structure and domain logic is distributed among the methods of the classes with their superclasses; effectively fragmenting the bowl of spaghetti into a dish of noodles. The system as a whole is nowhere to be seen.
3 SOME FUNDAMENTAL CONCEPTS AND THEIR USE IN BabyUML

3A The object

The notion of objects was introduced by Nygaard and Dahl with the Simula language [Simula]. The concepts were considerably refined in the Smalltalk language and run time system. [Smalltalk].

Fig. 4: The object

Objects are entities that encapsulate state and behavior. In this chapter, we use Smalltalk’s pure object model as illustrated in figure 4. The state of an object is defined by the values of its instance variables. Its behavior is defined by its methods. Neither state nor behavior is directly visible from outside the object; they can only be accessed through messages to the object. A message is intention-revealing; it specifies what is required, but not how this is to be produced ¹. When an object receives a message, it looks up a message dictionary to find the appropriate subprogram for handling the message. This subprogram is called a method. A method can read and change the value of the instance variables, and it can send messages to itself or other objects. Different objects can thus handle identical messages in entirely different ways².

Section 2B demonstrated our need for some sort of object clustering in order to inject some order into the noodles. The instance variables are in themselves less than useful. Some of them may point to what can be considered peers in a cluster, e.g., an activity predecessor. Some of them may point out of a cluster, e.g., from an activity to its resource. And some of them point to sub-objects that may be considered as parts of the object itself, e.g., from an activity to its name. Well chosen variable names can help a knowledgeable reader understand the semantics, but it is a weakness that we only see the structure from the perspective of a single node, we do not see the structure as a whole.

3B The class

In most object oriented languages, an object is an instances of a class. The class defines all features that are common to the instances of the class, notably their methods and the specification of their instance variables. Note that the class was not mentioned in the above description of the BabyUML object model. It is the object that holds the state and the methods are executed in the context of the

¹.This in contrast to a procedure call that uniquely identifies the procedure body.

².The Smalltalk notion of a messages keyword is similar to the UML Signal and an actual message is similar to the UML Signal instance.
object. The class is convenient for efficiency in space and time, but it is not an essential part of the object semantics.

A class inherits all the features of its superclass, it can add features of its own, and it can override methods defined in the superclass. A class with its superclasses can always be flattened into a single class with no superclass. This means that the actual distribution of features between the superclasses do not in any way influence the semantics of the object. I see this specialized form of inheritance as a device for code saving and code sharing, and do not overload the notion of a class with notions from philosophy and mathematics.

In BabyUML, classes and roles are complimentary. Classes are used to specify objects as white boxes, roles are used to describe them as black boxes in the context of ensembles of collaborating objects. More about roles in section 3G.

### 3C The Package

In UML, a package is used to group model elements. A package is a namespace for its members, and may contain other packages. A package can import either individual members of other packages, or all the members of other packages. In Java, similar packages are used to group classes and interfaces.

An object is an instance of a class. This class is a subclass of its superclass and so on up to the root of the class hierarchy. The classes in the superclass chain are typically members of different packages. An object is thus related to several packages, and the notion of a package is irrelevant in the context of interacting runtime objects.

### 3D The Component

The UML metaclass Component is a subclass of the Class metaclass:

“A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.

A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics). One component may therefore be substituted by another only if the two are type conformant. Larger pieces of a system’s functionality may be assembled by reusing components as parts in an encompassing component or assembly of components, and wiring together their required and provided interfaces.”

A babyComponent is an object that encapsulates other objects. It looks like a regular object seen from its environment and is characterized by its provided interface. Regular objects and components can be used interchangeably. Inside a component, we find a bounded structure of interacting Member Objects.

Figure 5 illustrates how the spaghetti of fig. 3 is replaced by a simple structure of three interacting components. Like UML, the notion of a BabyUML Component is recursive; I can organize several hundred thousand objects in a component structure.
so that I can deal with a manageable number at each level.

**Fig. 5:** A Component is an object that encapsulates other objects.

There are many advantages of an architecture based on the BabyUML components:

- My brain can better visualize how the system represents and processes information. The code includes the component specification; the code thus documents the high level system architecture.
- The notion of components makes it easier to ensure correspondence between the user’s mental model and the model actually implemented in the system.
- The component boundary forms a natural place to put a firewall for security and privacy. Indeed, it is hard to see how privacy and security can be achieved without some form of strict, object based component architecture.

The notion of a BabyComponent is useful in many context. A specialization is the DCA component described in section 5.

### 3E The Database

An early idea for system structuring was the idea of separating state and behavior. The first, 1963 version of our Autokon system was structured as a number of application programs arranged around a central data store that held information about the ship, its geometry and the arrangement of its parts. Different applications accessed the store through special access routines that transformed the store’s data structure to an apparent structure suitable for the application as illustrated in figure 6.

**Fig. 6:** Separating data and procedure.

This separation of state and behavior is very promising for our purposes. An object encapsulates state and behavior. The DCA paradigm encapsulates all of fig. 6 within a “monster object”. The data store becomes a “micro Database” that represents the object state. The access routines become Collaborations that view the Data in different perspectives. The applications become Algorithms that encode the object behavior. The DCA paradigm is described in more detail in section 5.

### 3F The Subroutine

The closed subroutine was invented by D. J. Wheeler in the summer of 1949 [Wilkes-76] in the EDSAC group in Cambridge, England. The group quickly developed a discipline of programming
where a main program called a number of subroutines; these subroutines again would call other subroutines and so on to an arbitrary depth. New insights spread quickly in those days. In 1963 in Oslo, we deployed Autokon, a system for the computer-aided design and manufacture of ships [Ree-HiNC]. One of its applications was a 50,000 line assembly program. It was structured as a hierarchy of subroutines as illustrated in figure 7, where each routine was documented on a single page flow-chart. This algorithmic decomposition gave us mastery over the program; it was so simple that there were obviously no deficiencies. (The subroutines themselves were not similarly structured until the advent of structured programming in the late sixties. )

Subroutines become methods in the world of objects. But methods are encapsulated in objects, so the simple notion of static algorithmic decomposition can only be applied within the object. The dynamic binding between message and method invalidates the notion of static algorithmic decomposition at the level of object interaction. This means that the old calling tree illustrated in figure 7 is broken at the object boundary because different objects may invoke different methods for the same message. It also means that breaking down a system on the subroutines is of limited value. The BabyComponents is a better candidate; the breakdown is then on state and behavior simultaneously.

3G The Role Model

Prokon/Plan [Ree-IFIP] was a project that attempted to create a system of planning and control in the shipbuilding industry based on the idea of negotiation. Line managers negotiate to achieve a mutually acceptable plan. The negotiation takes place in a computer system where each manager is represented by an object that acts on the manager’s behalf. The manager owns his objects and specifies the algorithms, including the criteria for when the object shall refer back to its owner for a manual decision. The idea is illustrated in figure 8.

Decisions about a new plan is made at the people level. The automated negotiation takes place on the black box level, i.e. where we only see the external properties of the objects. The negotiation algorithms are seen on the black box level while the algorithms controlling the behavior of individual objects are on the white box level. The black box level belongs to the shipyard as a whole, while the white box level belong to individual managers. This means that each manager decides on how to plan his own operations and also what he wants to plan manually and what he wants to automate. We
don’t care how you do it, but your objects must behave properly in the enterprise-wide community of interacting objects.

![Prokon sequence diagram](image)

The Prokon interaction diagram in [Ree-IFIP] showed specific objects, their links and interactions as shown in figure 9. Trapezoids represented concrete objects and arrows indicated concrete messages with their semantics described in the text. The diagram described a specific execution trace; notice that the **RRC** object occurs twice.

An attempt at implementing our ideas in Simula [Simula] failed because the Simula typing system insisted on our knowing the class of the objects and we missed language constructs for programming on the black box level.

The Prokon project lost its funding and died, but the vision has stayed with me. The transition to Smalltalk was a major step forward because its dynamic typing lets me focus on object interaction independently of classes and class hierarchy. The MVC is discussed in section 4. It was a result of thinking in terms of objects rather than classes, but there was still no construct for explicitly programming on the black box level.

The experience with MVC lead me to search for a new abstraction that let me work explicitly on the black box level. The result was *role modeling*, an abstraction that describe how a structure of objects interact to accomplish some desired functionary. Each object has a specific responsibility in an interaction\(^1\), we say that it plays a specific *role*. My book, *Working with Objects* [Ree-working], describes role modeling in depth and a theory of role modeling is given in Egil P. Andersen’s doctoral thesis [Andersen-97].

The notion of a role corresponds closely to the notion of an *artifact* as it is described by Steven Pinker in *How the mind Works* [Pinker-97]:

> What is an artifact? An artifact is an object suitable for attaining some end that a person intends to be used for attaining that end. The mixture of mechanics and psychology makes artifacts a strange category. Artifacts can’t be defined by their

\(^1\)More about responsibility driven design and roles in [Rebecca-03]
Artifacts come with being human. We make tools, and as we evolved our tools made us. One-year-old babies are fascinated by what objects can do for them. They tinker obsessively with sticks for pushing, cloth and strings for pulling, and supports for holding things up. As soon as they can be tested on tool use, around eighteen months, children show an understanding that tools have to contact their material and that a tool’s rigidity and shape are more important than its color or ornamentation. Some patients with brain damage cannot name natural objects but can name artifacts, or vice versa, suggesting that artifacts and natural kinds might even be stored in different ways in the brain.

An object interaction results in a desired result. An object that participates in an interaction is an artifact that makes its contribution towards this end; it plays its role in the interaction. The idea of role modeling has made it into UML under the name of Collaborations and Interactions.

As a role model example, we will consider the Observer Pattern as described in the book on Design Patterns [GOF]. A design pattern describes a solution to a general problem in such a way that it can be realized in many different ways and made to fit under many different circumstances. The Observer Pattern is described in the book with text, a class diagram, and a kind of collaboration diagram. We will here describe it with a role model.

The essence of object orientation is that objects collaborate to accomplish some functionality. Three questions need to be answered: What are the roles? How are they interlinked? How do they interact? The answer to the first two questions is the set of roles that work together to realize the functionality and the links between them. This is called a Collaboration in UML terminology.

As a role model example, we will consider the Observer Pattern as described in the book on Design Patterns [GOF]. A design pattern describes a solution to a general problem in such a way that it can be realized in many different ways and made to fit under many different circumstances. The Observer Pattern is described in the book with text, a class diagram, and a kind of collaboration diagram. We will here describe it with a role model.

The essence of object orientation is that objects collaborate to accomplish some functionality. Three questions need to be answered: What are the roles? How are they interlinked? How do they interact? The answer to the first two questions is the set of roles that work together to realize the functionality and the links between them. This is called a Collaboration in UML terminology.

As a role model example, we will consider the Observer Pattern as described in the book on Design Patterns [GOF]. A design pattern describes a solution to a general problem in such a way that it can be realized in many different ways and made to fit under many different circumstances. The Observer Pattern is described in the book with text, a class diagram, and a kind of collaboration diagram. We will here describe it with a role model.

The essence of object orientation is that objects collaborate to accomplish some functionality. Three questions need to be answered: What are the roles? How are they interlinked? How do they interact? The answer to the first two questions is the set of roles that work together to realize the functionality and the links between them. This is called a Collaboration in UML terminology.

As a role model example, we will consider the Observer Pattern as described in the book on Design Patterns [GOF]. A design pattern describes a solution to a general problem in such a way that it can be realized in many different ways and made to fit under many different circumstances. The Observer Pattern is described in the book with text, a class diagram, and a kind of collaboration diagram. We will here describe it with a role model.

The essence of object orientation is that objects collaborate to accomplish some functionality. Three questions need to be answered: What are the roles? How are they interlinked? How do they interact? The answer to the first two questions is the set of roles that work together to realize the functionality and the links between them. This is called a Collaboration in UML terminology.

As a role model example, we will consider the Observer Pattern as described in the book on Design Patterns [GOF]. A design pattern describes a solution to a general problem in such a way that it can be realized in many different ways and made to fit under many different circumstances. The Observer Pattern is described in the book with text, a class diagram, and a kind of collaboration diagram. We will here describe it with a role model.

The essence of object orientation is that objects collaborate to accomplish some functionality. Three questions need to be answered: What are the roles? How are they interlinked? How do they interact? The answer to the first two questions is the set of roles that work together to realize the functionality and the links between them. This is called a Collaboration in UML terminology.

As a role model example, we will consider the Observer Pattern as described in the book on Design Patterns [GOF]. A design pattern describes a solution to a general problem in such a way that it can be realized in many different ways and made to fit under many different circumstances. The Observer Pattern is described in the book with text, a class diagram, and a kind of collaboration diagram. We will here describe it with a role model.

The essence of object orientation is that objects collaborate to accomplish some functionality. Three questions need to be answered: What are the roles? How are they interlinked? How do they interact? The answer to the first two questions is the set of roles that work together to realize the functionality and the links between them. This is called a Collaboration in UML terminology.

As a role model example, we will consider the Observer Pattern as described in the book on Design Patterns [GOF]. A design pattern describes a solution to a general problem in such a way that it can be realized in many different ways and made to fit under many different circumstances. The Observer Pattern is described in the book with text, a class diagram, and a kind of collaboration diagram. We will here describe it with a role model.

The essence of object orientation is that objects collaborate to accomplish some functionality. Three questions need to be answered: What are the roles? How are they interlinked? How do they interact? The answer to the first two questions is the set of roles that work together to realize the functionality and the links between them. This is called a Collaboration in UML terminology.

As a role model example, we will consider the Observer Pattern as described in the book on Design Patterns [GOF]. A design pattern describes a solution to a general problem in such a way that it can be realized in many different ways and made to fit under many different circumstances. The Observer Pattern is described in the book with text, a class diagram, and a kind of collaboration diagram. We will here describe it with a role model.

The essence of object orientation is that objects collaborate to accomplish some functionality. Three questions need to be answered: What are the roles? How are they interlinked? How do they interact? The answer to the first two questions is the set of roles that work together to realize the functionality and the links between them. This is called a Collaboration in UML terminology.

As a role model example, we will consider the Observer Pattern as described in the book on Design Patterns [GOF]. A design pattern describes a solution to a general problem in such a way that it can be realized in many different ways and made to fit under many different circumstances. The Observer Pattern is described in the book with text, a class diagram, and a kind of collaboration diagram. We will here describe it with a role model.

The essence of object orientation is that objects collaborate to accomplish some functionality. Three questions need to be answered: What are the roles? How are they interlinked? How do they interact? The answer to the first two questions is the set of roles that work together to realize the functionality and the links between them. This is called a Collaboration in UML terminology.
"play this role at a certain time and in a certain context".

Figure 11 specifies how the objects interact when synchronizing subject and observer. The participants in the interaction are named after the roles they play in the interaction. We see that an inputter sends the `setState()` message to the subject, presumably changing its state. The subject then sends an `update()` message to all objects that, in this context, play the observer role.

![Fig. 11: An Observer interaction](image)

Note that many objects may play the observer role, but we are only concerned with the objects that play the observer role in a particular occurrence of the interaction. Also note that the collaboration diagram in Design Patterns assumed that an object called aConcreteSubject simultaneously played both the inputter and the observer roles. This is perfectly permissible, but might not be a good idea in many situations.

A role modeling tool called OOram was put on the market, but the interest was not sufficient to sustain it as a product. The reason could be that I had not found a conceptual bridge between roles and classes. I have recently found this bridge; classes specify the properties of objects, while the objects playing a certain role is found by a query on a set of existing objects. The class is thus a static property of an object, while the role is dynamic. A class describes the inside of an object. A role describes an object as it is seen in an ensemble of interacting objects.

This dichotomy between role and class, between the static and the dynamic, is the foundation of BabyUML. The goal is a new discipline of programming where the two are balanced so as to give the programmer new capabilities and new leverage.

### 3H Aspect Oriented Programming

Some programming problems cannot easily be captured by procedural or object-oriented code because they cut across procedures and objects. Aspect-oriented programming was introduced to handle such cross-cutting aspects of the problem. Examples are aspects related to security and performance.

At a first glance, it seems that role models and object interaction can be such an aspect since they cut across object boundaries. A technology similar to AOP should be able to support methods that are defined for a particular role and thus shared among all objects playing this role independently of their actual class. There is an appealing symmetry here: A class defines methods that are common to all its instances. What if a role defines AOP-like methods that are common to all objects that play this role. Interesting thought for a future experiment.
How can we build a system that the user experience as an extension of his own brain? How can we put the user in the driver’s seat so that he can not only run the program but also understand and even modify its operation? How can we structure a system so that each user sees an image of the world that corresponds to his own conception of it?

MVC was first conceived as a means for giving human users control of the computer resources. MVC bridges the gap between the users’ mental models and the information stored in the computer. The idea is illustrated in figure 12.

Fig. 12: Bridge the gap between the user’s mind and the stored data.

The domain of my first MVC was shipbuilding. The problem was project planning and control as described in section 2A. A manager was responsible for a part of a large project. His department had its own bottlenecks and its own considerations for planning. Other departments were different; a pipe shop was very different from a panel assembly line which was again very different from a design office. How could each manager have his own specialized part of the planning system while preserving the integrity of the plan as a whole?

The answer was to replace the “dead” activity records in traditional, procedure oriented planning systems with interacting objects. The objects would represent their owners within the universe of interacting objects. The objects could be specialized according to the needs of their owners, yet they could all interact according to a common schema.

I implemented the first MVC while being a visiting scientist with the Smalltalk group at Xerox PARC [Ree-MVC-1], [Ree-MVC-2]. The conventional wisdom in the group was that objects should be visible and tangible, thus bridging the gap between the human brain and the abstract data within the computer. This simple and powerful idea failed for the planning systems for two reasons. The first was that a plan was a structure of many activity and resource objects so that the focus on a single object at the time was too limiting. The other was that users were familiar with the planning model and were used to seeing it from different perspectives. The visible and tangible object would get very complex if it should be able to show itself and be manipulated in many different ways.

4A The Model

The terms data and information are commonly used indiscriminately. In the stone age, IFIP defined them precisely in a way that I still find very fruitful when thinking about the human use of computers [IFIP].
DATA. A representation of facts or ideas in a formalized manner capable of being communicated or manipulated by some process.

Note: The representation may be more suitable either for human interpretation (e.g., printed text) or for internal interpretation by equipment (e.g., punched cards or electrical signals).

INFORMATION. In automatic data processing the meaning that a human assigns to data by means of the known conventions used in its representation.

Note: The term has a sense wider than that of information theory and nearer to that of common usage.[IFIP]

So the user’s mental model is information, information as defined does not exist outside the human brain. But representation of information can and do exist outside the brain. It is called data. In the example, the Model is the data representing the activity network and the resources.

The Model data may be considered latent because they need to be transformed to be observable to the user and related to the user’s mental project model. We will discuss the Java implementation of the Model and its links to the View-Controller pair in section 5.

4B The View

The View transforms the latent Model data into a form that the human can convert into information as illustrated in figure 13.

The MVC triad could be implemented in a single class. If this class turns out to be complex, it can be simplified by a separation into separate classes. And if the users need to see the model data in different perspectives, then separating out several view classes is almost necessary.

Fig. 13: The View couples model data to the information in the user’s brain so that they appear fused into one.

I will discuss the Java implementation in section 5E on page 25.

4C The Controller

The Controller is responsible for creating and coordinating a number of related Views. I sometimes think of the Controller-View combination as a Tool that the user employs to work with the system’s latent information.1, 2

1. Note that the Smalltalk 80 Controller is responsible for input and thus different from the one discussed here.
2. Also note that some so-called MVC structures let the controller control the user interaction and thus, the user. This whole idea is fundamentally different from MVC as described here. I want the user to be in control and the system appear as an extension of the user’s mind. In short, I want the “main program” of the interaction to be in the head of the user. In the alternative “MVC”, the computer is in control and the system appears as an enforcer of company procedures. In short, the “main program” of the interaction is in the computer.
Fig. 14: The Controller creates and coordinates multiple Views

Looking back to section 3G on role models, we realize that Model, View, and Controller are roles played by components or objects. Their classes are unspecified and irrelevant to the MVC paradigm.

4D The anatomy of the Java user interface code

The Java tool is shown in figure 15. We see that the tool is divided into 4 strips:

1. The top strip contains command buttons. They are not part of the MVC and will not be discussed further.
2. The second strip is an instance of class DependencyPanel; it is a view that shows the activities with their technological dependencies.
3. The third strip is an instance of class GanttPanel; it is a bar chart showing the time period for each activity.
4. The fourth strip is an instance of class ResourcePanel; it shows the activities that are allocated to the resource in each time period.

Fig. 15: The anatomy of the MVC Java tool. (: ButtonStrip means an instance of class ButtonStrip).

An overview of the implementation is shown in the class diagram of figure 16. We see the classes described above and their main associations.

In my traditional programming style, the views would all be associated with the model. In this implementation, I reduce the number of associations in order to get a simpler and cleaner structure. The views are now subordinated the controller by being enclosed in a controller-managed component. This is indicated by a dashed line in figure 16. The Model and Controller are shown in heavy outline to indicate that they are the main collaborators in this implementation. The Views, being subordinate in this implementation, are shown in light outline. The Java library superclasses are shown dashed along the top of the diagram.
The component structure is reflected in the package structure as illustrated in figure 17.

We will go into the details when we discuss the model internals and system behavior in section 5.

**4E Controller code coordinates selection**

We will now take the selection function as an example of how the controller coordinates the behavior of the views.

Figure 18 shows the tool after the user has clicked on any of the actC activity symbols. The key to simplicity and generality is that the view being clicked only reports this event to the controller object. The controller decides that this is indeed a selection command, and that it shall be reflected in the appearance of all activityViews. This behavior is illustrated in the sequence diagram of figure 19.
The `inputView` role is shown to be some instance of class `ActivityView`. We see from figure 16 that `ActivityView` is an `awt.Button`, so it sends an `actionPerformed` event to its `actionListener`. All `activityView`s are created to let their `actionListener` be the `controller`:

```java
Controller>>public boolean isSelected(Activity act) { (Java 16)
    return (selection == act); (Java 17)
}
```

**Discussion.** The Observer pattern, also known as Dependents, Publish-Subscribe, or Changed-update, defines a one-to-many dependency between objects so that when one object (the subject) changes state, all its dependents are notified and updated automatically.\(^\text{[GOF]}\)

A variant of the selection interaction could use the Observer pattern to let the controller alert the views about a changed selection. On the face of it, this is very flexible, extensible, and so on. But in this case, it would merely be an obfuscator. The observer pattern is useful when the subject should be decoupled from its dependents. But here, the controller knows its views since it created them. The direct solution used here is the simplest and does not restrict flexibility and extensibility.

**Discussion.** We see from figure 16 on page 19 that the `controller` knows both `panels` and `activityViews`. An alternative could let the `controller` know the `panelViews` only. Each `panelView` could then act as a local `controller` for its `activityViews`. The programmer of the top level `controller` would then need to know the inner workings of the `panels`. I did not choose this solution because the responsibility for activity selection is anchored in the top level `controller`.

The `ActivityView` now asks the controller if it is selected and then repaints itself appropriately:

```java
ActivityView>>public void selectionChanged() {
    if (controller.isSelected(activity)) {
        setBackground(activity.color().darker()); (Java 11)
    } else {
        setBackground(activity.color()); (Java 13)
    }
}
```

and

```java
Controller>>public void actionPerformed(ActionEvent e) { (Java 1)
    ActivityView source = (ActivityView)e.getSource(); (Java 2)
    selection = source.activity(); (Java 3)
    for(ActivityView view : activityViews) { (Java 4)
        view.selectionChanged(); (Java 5)
    }
    repaint(); (Java 7)
}
```
5 DCA: THE DATA-COLLABORATION-ALGORITHM PARADIGM

We now come to the Model. Seen from the Controller, it looks like an ordinary object. But a single object that represents all activities and resources would be a monster, so we have to give it some structure. The DCA paradigm tells us how to master this monster object. The Model becomes a DCA component. It looks like a regular object from the outside, characterized by its provided operations. Inside, there is a well ordered and powerful object structure partitioned into three parts, Data, Collaborations, and Algorithms:

- **The D stands for Data.** Like any object, a DCA component encapsulates state and behavior. The state is a collection of domain objects that constitute the component’s base data. Their organization is declared as a number of relations in first normal form, guaranteeing data integrity. The data declaration can be read and understood independently of the system around it; an important step towards system simplicity. The recursive nature of components ensures that the data declaration is also recursive.

- **The C stands for Collaboration.** A DCA collaboration is useful when the schema for the base data is unsuitable for users of the data. Users then access the domain objects through the roles these objects play in the particular usage. Collaborations bridge the gap between usage and data by dynamically binding roles to domain objects. A role can be seen as indirectly addressing one or more domain objects, making it possible to address different objects at different times without changing the using code. The notion of Collaborations is derived from the OOram role model[1][Working] and corresponds to the external views used in database technology.

- **The A stands for Algorithm.** Algorithms occur in two places in the DCA paradigm. Some are local to the domain objects and are coded as methods in the domain classes. Other algorithms describe domain object interaction and is a property of the inter-object space. The interaction algorithms are coded in separate classes, distinct from the domain classes. This ensures that object interaction is specified explicitly and makes it easier to check the code for correctness and to study its operation.

5A The Model as a single object

The Java tutorial[1] describes an object as a number of fields (state) surrounded by methods (behavior) as illustrated in figure 20(a). This is actually a better illustration of the Smalltalk object than the Java object. In Smalltalk, the fields (“instance variables”) are invisible from outside the object; all access has to be through its methods. The Java object is different; the fields are visible from the outside. I write `x = foo.fieldX;` to access a field directly, and I write `x = foo.getFieldX();` to access it through a method.

Figure 20(b) shows an object model that we use as a starting point for discussing DCA. Borrowing terminology from UML, we use the term **owned attributes** to denote the state (fields, instance variables). We use the UML term **derived attributes** to

denote attributes that are computed rather than stored. For example, a *person* object could have *birthDate* as an owned attribute, while *age()* could be a derived attribute. Other methods implement the object’s provided operations.

Fig. 20: (The regular object is an instance of a class.
(a) The object as depicted in the Java tutorial. (b) A more accurate object model.

5B The DCA Component; a well-structured monster object

There are (at least) two definitions of the term *object*. The simple one is that *an object is an instance of a class*. The other one is that an object is an entity that encapsulates state and behavior. The two definitions are synonymous in a Java program, but they are different in DCA.

Fig. 21: The DCA component.

- **Data.** This corresponds to the owned attributes of the regular object. The fields are replaced by a “baby database” that holds the component’s domain objects and their structure. The term “database” is used in a restricted sense, it is a set of domain objects organized according to a conceptual schema. We do not assume persistence, concurrency, access control, security, or any other goodie usually associated with databases. We just call it the *Data*.
  - The Data is organized as a number of relations in first normal form, ensuring referential integrity.
  - The values of these relations are the domain objects (including structure objects).
  - The structure is represented in explicit relations. Contrast with my traditional representation where structure information is distributed among the domain objects. The DCA domain objects are correspondingly simplified.
  - The code for the Data part should ideally be declarative in the form of a conceptual schema, but I rely on defining some Java classes with only *getter* and *setter* methods in the current Java experiment.

- **Collaboration.** This corresponds to the derived attributes of the regular object. The Data conceptual schema may not be ideal for the access requirements of the different uses of the data. The collaborations implement external schemas, each optimized for a particular usage of the Data.
  - In UML[^UML], “a Collaboration describes a structure of collaborating elements (roles), each performing a specialized function,”
which collectively accomplish some desired functionality. Its primary purpose is to explain how a system works and, therefore, it typically only incorporates those aspects of reality that are deemed relevant to the explanation. Thus, details, such as the identity and precise class of the actual participating instances are suppressed.

Also “a CollaborationUse represents the application of the pattern described by a collaboration to a specific situation involving specific classes or instances playing the roles of the collaboration.”

- In this experiment, a DCA Collaboration is coded as a class that has the collaboration roles as attributes and database queries as its methods. In its abstract form, it is one or more temporary relations that are derived from the base relations by queries.
- The DCA Collaboration is an instance of the above class where the results of the queries are assigned to the role fields, thus binding roles to actual domain objects. A binding is valid in a certain context and at a certain time. It is a kind of dynamic, indirect addressing.
- Objects using a Collaboration see the base data in a perspective optimized for their needs. Note that these user objects can be internal or external to the Model.

Algorithm. The domain objects interact in order to “collectively accomplish some desired functionality”. Traditionally, the code for this interaction is implicit by being distributed among the domain objects. In the DCA paradigm, the code controlling the interaction is pulled out and centralized in the component’s Algorithms. Thus, a DCA Algorithm defines how the system accomplishes some desired functionality.

5C The MVC Model as a DCA component

The Model part of the network Java example is implemented as a DCA component. Some important objects are shown in figure 22. For illustrative purposes, the Data is separated into two sub-parts. The netBase holds the activity network in two relations: activities and dependencies, and the resourceBase has a single relation, allocations. (allocations has two attributes: week and activity).

The GUI is split into a controller object and three panelView objects, each with a layout algorithm that creates its display. In addition, the frontload command button activates the frontload and resourceAllocation Algorithms. All five Algorithms are users of the DCA Data and access it through a suitable Collaboration.

In the following, we will discuss the code for the dependencyPanel and the frontload Algorithm together with their tailored data access collaborations as illustrated in figure 22.

---

1. The DCA Collaboration corresponds to the UML CollaborationUse. My choice of name reflects my focus on objects rather than code.
5D The Data structure defined by a schema

The databases are defined by their schemas. Figure 23 shows the netBase schema expresses as a UML class diagram.

The corresponding Java class declarations are as follows:

```java
public class NetBase extends Observable implements Observer{
    private Set<Activity> activities = new HashSet<Activity>();
    private Set<MemberDependency> dependencies = new HashSet<MemberDependency>();
    ...
}
```

```
public class Activity {
    private Integer earlyStart, earlyFinish, duration;
    private String name;
    private Color color;
    ...
}
```

```
public class Dependency {
    private Activity predecessor, successor;
    ...
}
```
The complete Java code can be found at [http://heim.ifi.uio.no/~trygver/2006/09-JavaZone/](http://heim.ifi.uio.no/~trygver/2006/09-JavaZone/).

5E Code Example 1: Panel layout

Figure 24 illustrates that the unit on the horizontal axis in the **DependencyPanel** is the activity **rank**; i.e., the max length of the activity’s predecessor chain from the activity to the start of the network. Activities having the same rank are stacked vertically.

The **DependencyPanel** layout Algorithm is as simple as can be. (Actually too simple, it will often lead to overlapping dependencies.)

```java
private void addActivityViews() {
    Integer gridX = getSize().width / (rankedCollab.maxRank() + 1);
    Integer gridY = getSize().height / rankedCollab.maxSizeActivitySets();
    Integer x0 = 10;
    Integer y0 = 10;
    Dimension buttonExtent = new Dimension(gridX-50 , gridY-20);
    for (int rank=0; rank <= rankedCollab.maxRank(); rank++) {
        Integer xPos = x0 + (gridX * rank);
        Integer yPos = y0;
        for (Activity act : rankedCollab.activityListAtRank(rank)) {
            ActivityView actView = new ActivityView(controller, act, 24);
            activityMapActivityViews.put(act, actView);
            actView.setBounds(xPos , yPos , gridX-50 , gridY-20);
            controller.addActivityView(actView);
            add(actView);
            yPos = yPos + gridY;
        }
    }
}
```

This layout Algorithm accesses the activities through the **rankedCollab**, an instance of the **RankedCollab** class. This collaboration presents the data in a table with two columns: rank and activity. The table is accessed through the call to **activityListAtRank()** in [Java 53] above.

The corresponding code in the **RankedCollab** class is as follows:

```java
public List<Activity> activityListAtRank(Integer rank) {
    List<Activity> activityListAtRank = new ArrayList<Activity>();
    for (Activity act : netBase.activities()) {
        if (rankOf(act) == rank) {
            activityListAtRank.add(act);
        }
    }
    return activityListAtRank;
}
```
```java
activityListAtRank.add(act);
}
}

private Integer rankOf(Activity act) {
    // Extremely inefficient. Early candidate for cashing.
    // NOTE: A feature of the structure, not an individual activity
    Integer rnk = 0;
    for (Activity pred : predecessorsOf(act)) {
        rnk = Math.max(rnk, (rankOf(pred))+1);
    }
    return rnk;
}
```

The complete RankedCollab code can be found at

5F Code Example 2: Frontloading

Frontloading is the calculation of the earlyStart and earlyFinish for each activity given the start time of the start activities. We see from figure 24 that actA and actB can both start when the project starts, e.g., in week 1. actA then finishes in week 2 and actB in week 7. We can now compute for earlyStart and earlyFinish for actC. actD can now be computed since we know the earlyFinish for both actC and actB. The result of the frontloading is shown in the gantt diagram of figure 2.

The frontloading operation is traditionally distributed among the activity objects. The default method could look like the following:

```java
Activity >> public void frontloadSimple (Integer startWeek) {
    earlyStart = startWeek;
    for(Activity pred : predecessors()) {
        earlyStart = Math.max(earlyStart, pred.earlyFinish() + 1);
    }
}
```

There are problems with this simple solution. The method cannot be triggered in an activity object before the earlyFinish of all predecessors are known. This means that the frontload operation belongs in the inter-activity space and must be treated at a higher system level.

The common frontload logic could be in a method in the Model class, but I feel that this is an overloading of a class that should be clean and simple. So the top level frontload() method is coded in a separate class, the FrontloadAlgorithm class. Three problems need to be resolved. One is identifying activities that are ready to be frontloaded. The second is to determine the earlyStart of an activity once all its predecessors have been loaded. The third is to compute the earlyFinish for an activity once its earlyStart is known.

1. Identifying activities that are ready to be planned is essentially a query on the Data objects. This work properly belongs in a collaboration class, here the FrontloadCollab class.
2. The earlyStart of an activity depends on all its predecessors and all modifiers such as activity overlap etc. This logic belongs in the inter-activity
space and is here coded in the FrontloadAlgorithm class.

3. The earlyFinish of an activity once its earlyStart is known depends on the activity alone. The code, therefore, belongs in the Activity class.

We’ll discuss the resolution of these actions in turn.

5F1 FrontloadCollab, the frontloading collaboration

There is a simple solution for finding the activities that are ready to be loaded. Activities with \( \text{rank}=0 \) have no predecessors, so they can be loaded first. Once they are done, activities with \( \text{rank}=1 \) can be loaded, and so on. This solution has the added benefit that we could reuse the RankedCollab collaboration.

I have chosen a different solution here because I am not 100% certain that the ranking solution holds for all possible structures. More important, I choose a query-based solution to illustrate how a query result changes through the frontloading process. Here is a query in an unspecified language that finds a candidate activity for frontloading:

```sql
define frontloader() as
  (select act
   from Activities act
   where act.earlyStart == null and ( for all pred in predecessors(act): pred.earlyStart != null )
  ) someInstance
```

Here is an excerpt from the FrontloadCollab class with the corresponding Java code:

```java
public Activity frontloader() {
    for (Activity act : netBase.activities()) {
        if (act.earlyStart() == null) {
            Set<Activity> predSet = predecessorsOf(act);
            if (areAllDone(predSet)) {
                frontloader = act;
                return(frontloader);
            }
        }
    }
    return null;
}
```

The areAllDone() method called from line (Java 74) is shown below:

```java
private boolean areAllDone(Set<Activity> actSet) {
    boolean allPredsDone = true;
    for (Activity pred : actSet) {
        if (pred.earlyStart() == null) {
            allPredsDone = false;
            break;
        }
    }
    return allPredsDone;
}
```
The `FrontloadCollab` code is not trivial. But it is nicely isolated. I can give it to a colleague and ask her to audit and sign it. (So that an unlikely bug will be her fault, not mine.) The complete code for class `FrontloadCollab` can be found at 


---

5F2 `FrontloadAlgorithm`, implementing the frontloading interaction

The frontloading interaction is implemented in the `FrontloadAlgorithm` class. It is here the simple, default case with no modifiers or other obfuscations:

```java
public void frontload(Integer startWeek) {
    // reset all
    for (Activity act : frontloadCollab.resetters()) {
        act.setEarlyStart(null);
    }
    // frontload all
    Activity frontloader;
    while ((frontloader = frontloadCollab.frontloader()) != null) {
        Integer earlyStart = startWeek;
        for (Activity pred : frontloadCollab.frontPredecessors()) {
            earlyStart = Math.max(earlyStart, pred.earlyFinish() + 1);
        }
        frontloader.setEarlyStart(earlyStart);
    }
}
```

This code is pure with no confusing side issues. It is thus a good starting point for dealing with more complex situations.
6 THE BabyIDE LABORATORY

The experiments reported in section 5 and section 6 has revealed the kind of high-level structures that shall be included in the BabyUML discipline of programming. The next major step is to experiment with the BabyIDE -- an integrated development environment for BabyUML. In the experiment, I try out novel semantics for classes and metaclasses together with tools for design, compilation, and inspection.

6A The Integrated Development Environment

Nusse, the first Norwegian computer anno 1953, was a computer were the smallest addressable unit was a word of 32 bits. Data and operations were indistinguishable in the computer memory; my programs typically modified themselves. I moved a word to the accumulator register and the computer treated it as an operand in an operation. I moved the same word to the operation register and the computer executed it. My mind-set was binary, my programs were written in binary, I ran and inspected the programs from the binary console. There was an exact correspondence between the program in my mind and the bits in the computer. Figure 25 illustrates the situation. The man-machine system was harmonious because the same conceptual framework applied throughout my thinking, coding, and inspecting (including debugging).

I moved to larger computers and higher level languages. A gap opened between my mind and the realities of the computer. I thought in FORTRAN, I coded in FORTRAN, a compiler translated my code into binary, but the inspect path remained binary. Harmony was lost; I have spent innumerable hours debugging my FORTRAN programs by manually decoding pages of hexadecimal dumps.

The loop in figure 25 was again closed when the plain compilers grew into integrated development environments. First for FORTRAN, today I use the Java NetBeans IDE\(^1\) and think, code, inspect, debug, and even refactor a program within the conceptual framework of Java.

I introduced a new mismatch when I began thinking in terms of MVC and DCA. I had to translate my mental models into Java code, and then compile, inspect, and debug within the Java environment. My mental model was in my head only, and a Java expert reading my code couldn’t possibly

\(^1\) http://www.netbeans.org/
guess my models. I could comment the code, but the comments would clutter the code and often be misleading. Extensive documentation could help, but I can never promise to maintain exact correspondence between documentation and code. I am highly motivated to improve the code; comments and documentation can be fixed later.

I tried using an advanced UML modeling tool for creating the demo program. There were several difficulties that hindered me working exclusively in UML. The three most important were:

- The tool only implemented parts of the UML 2.0 definition. The first stumbling block was that it lacked a necessary feature in the UML sequence diagram.
- The code generator was incomplete. The generated code was a mere skeleton; I had to fill in most of the code in Java. So much so that there was very little gain from using the additional tool and I quickly abandoned it.

These two difficulties can, in principle, be overcome with a more complete tool implementation. But the third is inherent in the idea of a model with a code generator:

- The code generator only transforms the model from UML to Java. I still had to inspect and debug in terms of Java. The correspondence with my MVC/DCA mental model was far from simple. Harmony is lost.

6B The BabyIDE Object Notation

The BabyIDE is centered around objects and their interaction. Interacting objects can only see the provided operations of their collaborators. Conceptually, an object appears to encapsulate state and behavior. In reality, the object is implemented by the object itself, and a class with its superclasses. BabyIDE introduces new notations for these perspectives. The encapsulated object notation is described in section 6B1. The conceptual object notation is described in section 6B2. The object implementation with class and superclasses is discussed in section 6C2.

The notation presented here symbolizes concrete objects with identity, state, and behavior. The notation will later be extended to denote roles; the artifact abstraction on objects described in section 3G.

6B1 The Encapsulated Object notation

An object is encapsulated; it can only be accessed through its provided operations. Its attributes and methods are invisible from its environment. BabyIDE uses the encapsulated object notation shown in figure 26 to denote an object seen as a black box.
The BabyIDE notation for an object is a rounded rectangle; its corners are rounded to distinguish it from the UML classifier. An object has a unique identifier, the objectID. It is shown in angle brackets: `<2232>`. Some objects have a name, this is then shown after the objectID. There are two equivalent notations. The inline form in figure 26 (a) is useful in simple diagrams. The compact form of figure 26 (b) uses the UML symbol for an interface. A tool can pop up the interface dynamically so as to save screen acreage.

Note that we use the Smalltalk syntax in this chapter. For example, `name` is equivalent to the Java `name()` method call. (In Java, `name` is a reference to the corresponding instance variable, while instance variables are invisible from outside the object in Smalltalk). The Smalltalk `frontload: firstWeek` is equivalent to the Java `frontload (firstWeek)`.

The encapsulated object notation is useful in “wiring diagrams” showing systems of interlinked objects as illustrated in figure 27.

**6B2 The Conceptual Object Notation**

An object encapsulates state and behavior. We all know that many of the object’s features are specified by its class with its superclasses, but do not let it confuse us. It is the object that has state and behavior, it is the object that interacts with other objects. So we hide the classes and pretend that the object itself holds everything of interest. The result is the conceptual object; very effective as a concept and very inefficient if naively implemented. We again use a rounded rectangle to denote an object and get the white box view of figure 28. We see selected object features, but we do not see how they are implemented. The conceptual object has three compartments:

- The top compartment shows the objectID `<2232>` together with a possible name.
- The middle compartment shows the names and values of the object’s attributes (instance variables).
- The bottom compartment shows the operations. A tool could also show the code of the corresponding method so that it can be inspected and edited.
The Smalltalk Virtual Stored Program Object Computer

I regard Smalltalk as the ideal proving ground for my experiments with BabyIDE. The Smalltalk notions of class, method, programming language and programming tools are all realized by objects in the image. A new class is created by sending the message `new` to its metaclass; another object. The code for a method is translated from its text form to byte codes by a compiler method that is part of the class object. This means that BabyIDE can implement its own notions of programs, programming languages and tools by simply replacing the Smalltalk library classes with alternate ones.

In this section, we give a detailed description of how programs are implemented in the default Smalltalk class library. We need a deep understanding of the default Smalltalk way in order build BabyIDE within and on top of it.

Figure 29 shows Smalltalk as a virtual, stored program, object computer. Object, because all data are represented as objects; even booleans, numbers, and characters; classes and methods; stacks and activation records; inspectors and debuggers. Virtual, because it is realized in software by the Smalltalk Virtual Machine (VM). Stored program because programs are represented as regular objects.

Fig. 29: The Smalltalk stored program virtual

6C1 The Smalltalk object computer implementation

Every object has an object identity, `objectID`, that is unique through space and time. There shall never be another object with the same `objectID` anywhere in the world. We can thus be sure that any object can be linked to any other object without any danger of confusion.

Smalltalk objects are stored in its object space, the image. The VM creates a new object when told to do so by a method; returning the `objectID` of the new object so it can later be the receiver of messages.
The bytes representing an object are stored on the computer’s heap. It would be ridiculously inefficient if all the features of an object such as attributes and methods should be stored in every object. The object only stores its state values as indicated on a white background in figure 30. In addition, the object has a number of hidden values; the most important being its identity and a link to the class object. The rest is delegated to the class and superclass objects as illustrated in the figure.

The objects in figure 30 are as follows:

- The object that is in the center of our interest has objectID=<2232> and stores the values ‘actD’, 3, 14, 16, {<1267>. <1125>.}

- The <2232>actD object is an instance of class Activity, represented in the <0148>Activity object. This class object has a link to its superclass, <0002>Object. The superclass of <0002>Object is nil, thus terminating the superclass chain.

- The names of the object’s attributes are the union of the attributes attributes of the class and all its superclasses. They are shown in the middle portion of the conceptual object.

- The object’s operations are a union of the operations defined in the methodDict attribute of the class and all its superclasses. The operations are shown in the bottom portion of the conceptual object. A tool can display the corresponding methods.
It is interesting to notice that the `instanceOf` relation essentially defines the object’s semantics, while the exact distribution of feature definitions along the `subclassOf` relations is irrelevant to the object and thus is in the nature of a comment.

A class object has a `methodDict` attribute that binds operations (selectors) to the corresponding `CompiledMethod` objects. A `CompiledMethod` object contains a sequence of byte codes (VM instructions) and also a link to the corresponding source code. There are byte codes for getting and setting attribute values as well as for sending messages to specified objects.

Object behavior is activated when an object receives a message. The message is an object with attributes for sender, receiver, message selector (operation) and actual parameters. The VM locates the receiver’s class object and looks up its `methodDict` dictionary to find the corresponding `CompiledMethod`. If not found, it recursively tries the `methodDict` of the superclass. If this search fails, the VM starts the search anew with the default selector `doesNotUnderstand`. The search will never fail because the `doesNotUnderstand` method is defined in the root class. Once the search has succeeded, the VM creates an activation record (another object) and puts it on the stack. It then begins executing the method’s byte codes in the context of the activation record. The method’s byte codes can send a message to an identified receiver object, and the story repeats itself.

The `<2232>actD` object responds to the `frontload` operation. The corresponding method is stored in the `methodDict` attribute of the `<0148>Activity` class object. The `<2232>actD` object also responds to the `inspect` method that is stored in the `<0002>Object` class object. Both of them are visible to the collaborators of the `<2232>actD` object as bona fide operations on that object.

There are two important kinds of relationships in figure 30; the `instanceOf` and the `subclassOf` relationships. The implementation of an object consists of one `instanceOf` relation to its class object, followed by any number of `subclassOf` relations up the superclass chain. The two kinds of relations can easily be confused by the unwary.

For example, the features of the `<2232>actD` object are stored in attributes in its class object, `<0148>Activity` with its superclasses. Correspondingly, the features of the `<0148>Activity` class object are stored in attributes of its class object, its `metaclass` (not shown in figure 30). The “off by one” is here a potential source of extreme confusion. I look at an object, its features are in the attributes of its class object. A look at the class, its features are in the attributes of the metaclass. It is hard to see where the UML class symbol fits in. It does not symbolize the instance, because the instance is a merge of all classes in the superclass chain. It is does not symbolize the class object, because the features shown in the UML symbol are not features of the class object. The conclusion is that the UML class symbol represents the source code and is inappropriate for describing run time objects in a stored program object computer. The motivation for introducing the BabyIDE object notation discussed in section 3B was to avoid this confusion.

The class object `<0148>Activity` has its own provided operations with the corresponding methods.
In the default Smalltalk implementation, it responds to the message `compile:`. The corresponding method is a compiler that translates source code into a `CompiledMethod` and installs it into the `methodDict` for later execution by an instance of this class. In BabyIDE, I will use this feature to provide different compilers for different languages, new or old. The CompiledMethod has a link to its source code, thus making it possible to close the loop of figure 25.

### 6D The BabyIDE Laboratory Implementation

We argued for the choice of Smalltalk for implementing BabyIDE in section 6C. The choice of its Squeak dialect is harder. It is fairly easy to learn the semantics and syntax of the Smalltalk programming language, but it can be frustrating to become familiar with its pragmatics and class libraries. Squeak is even more frustrating because it is evolving very rapidly and includes a large number of undocumented features in different states of completion. But the advantages far outweigh the objections:

- The most important argument is that there is a very active and creative community around Squeak. Many ideas and even programs that are useful to BabyIDE are to be found in the Squeak program libraries and mailing lists.
- Squeak is open source; there are no obstacles to the distribution of the BabyIDE laboratory to anybody who might want to experiment with it.
- The Squeak VM is also open source; the program is written in a subset of Smalltalk and automatically translated to C. This means that it is feasible to modify the BabyIDE VM if necessary.

Figure 31 illustrates the BabyIDE implementation. The baby objects have their own classes, metaclasses, and metametaclasses; but they can freely interoperate with regular Smalltalk objects because they conform to the simple VM conventions.

### 6D1 The BabyIDE Layered Architecture

Classes and metaclasses come in pairs in regular Smalltalk. The class object holds the properties of its instances. The corresponding metaclass is needed to hold the features of the class object itself such as its static attributes and methods. Many Smalltalk novices find it hard to distinguish between regular and static attributes and methods. In BabyIDE, we remove the notion of static attributes and methods from the core classes. There is no loss of generality; we can always implement the notions of shared features at a higher abstraction level. The result is a set of clear core constructs that permit us explore new languages and tools for our new discipline of programming.
Every object is an instance of a class. In BabyIDE, this is implemented by every object having a link to its class object. The class is represented by an object that has a link to its class object, the metaclass. Finally, the metaclass object has a link to the metametaclass which is an instance of itself. This idea of a layered architecture is fundamental to BabyIDE semantics, but the exact number of layers depend on circumstances. The core layers from the concrete to the abstract are shown in figure 32:

**Fig. 32: The BabyIDE Instantiation Architecture**

**M3 - Non-class layer:** Here are the non-class objects, typically application and support objects.

**M2 - Class layer:** Here are the regular classes. Class objects create new instances, act as repositories for information common to these instances, and know how to translate code from a human form to executable binary.

**M1 - Metaclass layer:** Metaclass objects are class objects that have classes as their instances. They serve as repositories for the features that are common to their instances; i.e., a set of classes of the same kind. This ensures that BabyIDE is genuinely extendable because different sets of classes can have different compiles, inspectors, etc.

**M0 - MetaMetaclass layer:** There is a single object in this layer; it is called BMetaMetaclass. Directly or indirectly, all BabyIDE objects are instances of this class. BMetaMetaclass is an instance of itself so it had to be created by a somewhat tricky program.

Note that the BabyIDE layered architecture is an instantiation hierarchy, the implementation of the object with its class and metaclasses is shown in figure 33 1. The figure says that has a link to its class object, Activity. Like every other object, this object has a link to its class, MetaSimpleClass. In its turn, this objects has a link to its class again, Metametaclass, which is an instance of itself.

**Fig. 33: The example instantiation structure**

---

1. We use the UML «instance of» to link an object to the object representing its class.

---
strained by letting \texttt{MetaSimpleclass} be an instance of itself, thus terminating the instantiation chain. We want BabyIDE to be extensible with different kinds of classes and different IDEs. We therefore added an additional layer, the \texttt{MetaMetaclass} layer. BabyIDE can thus be extended by the addition of new metaclasses. Tool writers can leverage this power to create new concepts.

The orthogonal \textit{class inheritance hierarchy} is a very powerful device for code reuse and code sharing. Figure 34 shows the inheritance hierarchy of the current example. We see that it bears no relationship to the instantiation hierarchy of fig. 33. The inheritance hierarchy can be refactored without changing the system semantics. This particular solution is, therefore, relatively unimportant.

\begin{center}
\textbf{Fig. 34: The example inheritance structure}
\end{center}

\begin{center}
\begin{tikzpicture}
\node[draw,rounded corners] (O) {\texttt{Object}};
\node[draw,rounded corners] (B) [below of=O] {\texttt{Behavior}};
\node[draw,rounded corners] (A) [right of=B] {\texttt{Activity}};
\node[draw,rounded corners] (M1) [below of=A] {\texttt{MetaMetaclass}};
\node[draw,rounded corners] (M2) [below of=M1] {\texttt{MetaSimpleclass}};
\draw[->] (O) -- (B);
\draw[->] (O) -- (A);
\draw[->] (A) -- (M1);
\draw[->] (M1) -- (M2);
\draw[->] (B) -- (M1);
\end{tikzpicture}
\end{center}

The human mind is well equipped for understanding a hierarchy, but it finds it harder to handle two of them simultaneously. BabyIDE gains its power and extensibility from its four layered instantiation architecture, but application programmers should still only see the familiar class inheritance. The toolmaker can exploit the power of metaclasses to give the application programmer leverage through high-powered concepts and tools.

\textbf{6D2 Example implementation of the class layer}

Every object is an instance of some class. The class of a class object is a \textit{metaclass}. It is a rich source of confusion that the features of the \texttt{actD} object are implemented by the objects shown in fig. 30, while the features of the \texttt{Activity} class object are implemented by the objects shown in fig. 35.
Fig. 35: Implementation of the Activity class

Note that the Activity class object responds to its own messages such as new and compile. The corresponding methods are found in the class of the class, Metaclass (or actually the methodDict of its superclass, Behavior).

A toolmaker challenge is to exploit the power of a flexible instantiation structure without confusing the application programmer by forcing him to work with complex metaclasses.
7 CONCLUSION

The main thrust of the BabyUML project is to create a new discipline of programming where there is a balance between the static classes and the dynamic roles. Current IDEs such as NetBeans give the programmer excellent control over the specification of objects through defining their classes. Dynamic issues are neglected in current IDEs. Yet, the essence of object orientation is that objects interact to reach a desired result. The network example reported in section 4 and section 5 has served to exemplify how structures of interacting objects can realize the MVC and DCA paradigms.

I claim that the MVC and DCA paradigms often will lead to simpler and more readable code. MVC is long proven, DCA has as yet only been applied to the current simple Java problem. I believe both MVC and DCA are valuable as they stand and that it will be well worth while to use them in selected applications.

What remains to be done is to design and implement a programming environment that provides a balance between classes and roles. This will be the BabyIDE that will be built on top of the results reported in this chapter. Its top level architecture is compactly expressed in fig. 13 where a data model closely corresponds to the user’s mental model and a view bridges the gap between the abstract computer data and the user’s brain. Seen in this perspective, application programmers are the users of the tools to be created in the next experiment. The MVC and DCA paradigms will be the metamodels of the programmer’s perception of a program and the and the corresponding program descriptions in the computer. This will give added leverage that improves program readability and reduces program volume. MVC and DCA are but examples; BabyIDE shall be extensible so that it can support many different paradigms.

In my first BabyIDE experiment, I tried treating the UML metaclasses as genuine Smalltalk classes. I failed because there appeared to be a fundamental difference between the UML model of an object and the corresponding run time object. Take the notion of a link. In Smalltalk, it appears as the value of an instance variable. In UML, it appeared to be three interlinked instances: anAssociationEnd, anAssociation, anotherAssociationEnd. I did not find it worth while to continue this, somewhat naïve, approach.

The second experiment was to implement my own versions of class and method objects. This forced me to go deeply into the nature of objects and the fundamental difference between instantiation and generalization. The result of this experiment was the BabyIDE laboratory as reported in section 6. This laboratory forms a powerful and conceptually simple foundation for further development.

I tried to continue the second experiment by populating the BabyIDE laboratory with high level programming tools, but quickly realized that I couldn’t design and build the tools before I fully understood what they were to achieve, i.e., the interacting run time objects. The third experiment was done in order to create concrete examples of the desired
results of the initial BabyIDEs. This *activity net-
work* experiment was done in Java for two reasons.
One was to decouple IDE issues from the structures
themselves. The other was to communicate some
of the BabyUML ideas to a broader community.
The result was the MVC and DCA structures
reported in section4 and section5.

Most of my almost 50 years in computer program-
ning have been devoted to creating tools for peo-
ple. My success criteria have been the happy and
effective user rather than the weighty scientific
paper. The success criterion for the BabyUML
project is thus the happiness and effectiveness of
the programmer. The results of the next experiment
will be decisive.

This next and *fourth experiment* will be to return
to the BabyIDE laboratory and create high level tools
for programming and documenting systems that
follow paradigms such as DCA and MVC. I will
clearly need to harness imperative, algorithmic
programming as well as the declarative definition
of data structures. I will need class oriented pro-
gramming to define the nature of the objects and I
will need role modeling to define their interaction.
I will also need new debuggers and inspectors to
close the loop of figure 25. A great deal of pro-
gramming is clearly needed, and it is probably far
in excess of what can be achieved by a single pro-
gramer (me) working alone. So I hope that other
people will be inspired to pick up the loose ends
from my ideas and experiments to create new and
interesting results. There might even be an adven-
turous person who will join me in realizing the
BabyIDE dream.
8 REFERENCES.

<table>
<thead>
<tr>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 2314</td>
<td>IBM 2314 direct access storage facility. From IBM archives &gt; exhibits. [web link] [<a href="http://www-03.ibm.com/ibm/history/exhibits/storage/storage_2314.html">http://www-03.ibm.com/ibm/history/exhibits/storage/storage_2314.html</a>]</td>
</tr>
<tr>
<td>#</td>
<td>Reference</td>
</tr>
<tr>
<td>-------</td>
<td>---------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

The idea and implementation of the Smalltalk stored program object computer is due to Alan Kay, Dan Ingalls, Adele Goldberg and the Smalltalk team at Xerox PARC [Smalltalk]. UML is the combined result of a great number of people. Taken together, they have documented many concepts that are useful for modeling large systems of interacting objects [UML]. I am grateful to Dan Ingalls for helping me create my own class, metaclass, metametaclass and method objects in Squeak. Many thanks to Ragnar Norman for sharing his deep understanding of database technology and for helping me force my brain to think in declarative terms without immediately translating to my usual imperative style. (I apologize for any misrepresentations of his advice). My sincere thanks also to Johannes Brodwall for his intelligent support and advice on Java technology.