©2007 Nova Publishers. Posted with permission.
Preprint of chapter in
Computer Software Engineering Research; ISBN: 1-60021-774-5;
Nova Publishers; Hauppauge NY; 3rd Q. 2007

Programming with Roles and Classes: the BabyUML Approach

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 simplicity and leverage. Simplicity helps me to think clearly and a reader to understand and audit my code. Leverage lets me say more with less. The end result shall be a new interactive development environment with appropriate languages and tools for supporting high level abstractions.

The essence of object orientation is that objects interact to produce some desired result. Yet current programming languages are focused on individual objects as they are specified by their classes; there are no explicit language constructs for describing communities of interacting objects. In BabyUML, I will zoom back from the classes and let my code specify the roles that objects play in collaborations and interactions.

The BabyUML project is experimental; its ideas and concepts are explored and their feasibility demonstrated with actual code and running programs. One experiment is completed, it explores an old and a new paradigm for organizing objects in clear and explicit structures. The old is *MVC*, the Model-View-Controller paradigm that describes the objects bridging the gap between a human mental model and the corresponding data stored in the computer. The new is *DCA*, the Data-Collaboration-Algorithm paradigm where the collaborating objects are explicitly identified by the role they play in an interaction, and where the interaction pattern is explicitly defined in terms of these roles.

Another experiment shall lead to *BabyIDE*, an integrated development environment that exhibits a balance between classes and roles. BabyIDE will be part of a new discipline of programming where programmers can work consistently at a high conceptual level throughout coding, debugging, testing, and maintenance. It will be implemented in a corner of Smalltalk that I have called the *BabyIDE Laboratory*. In the last part of this chapter, I describe the laboratory and how it will support new programming paradigms and tools. I finally indicate the future direction towards a workable BabyIDE.

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 recorded computer bug. [1] This first bug was an "act of God"; most of the later bugs are blunders of our own making and the fight against them has been an essential part of software engineering ever since. The following quotes from the first NATO Software Engineering conference [2] could have been uttered today: [1]

> *<u>David and Fraser</u>: 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.*
>
> *<u>Dijkstra</u>: The dissemination of knowledge is of obvious value -- the massive dissemination of error-loaded software is frightening.*

The needs of society are still beyond us. An insatiable software market ever wants more, and we keep promising more than we can deliver. Major projects are delayed and even cancelled. Delivered software is buggy and hard to maintain. In his 1980 Turing Award lecture, Tony Hoare succinctly stated our choices [3]:

> *"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."[2]*

The other 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 our systems. But any given test method can only find a certain percentage of the all the 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: [3]

> *"Program testing can be used to show the presence of bugs, but never to show their absence!" [4]*
>
> *"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." [5]*

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

[1] Quoted with permission from NATO, CBP.

[2] ©ACM 1981. Quoted with permission.

[3] Quoted with permission from Hamilton Richards, Univeristy of Texas.

thinking, designing, coding, and maintenance. 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 any 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 tools for quite some time now, and Hoare's other way has been my way. I started the *BabyUML project* in an attempt to remedy this deplorable situation, hoping once again to experience the pleasure of following Hoare's first way. The goal of the project is to increase my confidence in my programs. The keywords are simplicity and leverage. Simplicity helps me to think clearly and a reader to understand and audit my code. Leverage lets me say more with less. The end result shall be a new interactive development environment with appropriate languages and tools for supporting high level abstractions.

The essence of object orientation is that objects interact to produce some desired result. Yet current programming languages are focused on individual objects as they are specified by their classes; there are no explicit language constructs for describing communities of interacting objects. In BabyUML, I will zoom back from the classes and let my code specify the communities with abstractions taken from OOram role modeling [6] and the concepts of collaboration and interaction from the OMG Unified Modeling Language®. [7]

The abstractions need to be represented as computer programs. I need new tools that bridge the gap between my brain and those programs. I want my code to be effectively chunked and self documenting so that other people can read it and grasp the system architecture and operation. 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. The BabyUML success criterion is that programmers shall be happier and more effective when they use its results. Programmer happiness is closely coupled with powerful concepts, responsive environments, exploration, evolution, and excellence.

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. [8] BabyUML is, somewhat whimsically, named after this computer because it is based on the idea of a *stored program object computer* such as it is pioneered in Smalltalk. [9] The other part of the name, *UML*[1], 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 my project.

Most of my almost 50 years in computer programming have been devoted to creating tools for people. My success criteria have been the happy and effective user rather than the weighty scientific paper. This chapter is an engineering status report on the project. Most of the chapter is about harnessing known principles for the purposes of the project. Some of the chapter is about new ideas; the most important are identified in the conclusion (section 7).

The BabyUML project is experimental because I need to use a tool in order to understand how to improve it. The result of the BabyUML series of experiments shall be a new discipline of programming that includes abstractions, processes, and computer tools. One or more new programming languages may or may not be required. I expect to find many useful concepts in UML. I do not commit to applying UML concepts correctly according to the specification, but will merely let them inspire my engineering solutions. One important simplification is that BabyUML is

[1] UML is a registered trademark of Object Management Group, Inc. in the United States and/or other countries.

limited to sequential programming while UML also caters for parallel processes.

In section 2A, I describe a simple example taken from activity network planning that will be used to illustrate the concepts presented in this chapter. In section 2B, I use this example to illustrate why my old programming style can fail when scaled up to large problems.

In section 3, I have selected some fundamental ideas that have proven their worth in the past and discuss them from a BabyUML perspective. Section 3A stresses that BabyUML see the *object* as an entity that encapsulates state and behavior; it can be more than a simple instance of a class. Section 3B describes the *class* as a descriptor of individual objects. Section 3C describes the *role model* or *collaboration*. This is an ensemble of objects that interact to realize certain functionality. A *role* is a link to an object that makes a specific contribution in a collaboration. The link is dynamic; it is only valid at a certain time and in a certain execution of a collaboration. The BabyUML project shall achieve its goal by creating BabyIDE, an interactive programming environment where there is a balance between the *classes* that describe what the objects *are* and the *roles* that describe what the objects *do* when they interact at runtime.

The chunking of run-time objects is critical to the mastery of large systems. Section 3D describes a *BabyComponent* as a "monster object" that looks like a regular object in its environment. This object is completely characterized by its provided interface and encapsulates *member objects* that are invisible from outside. Different components can structure their member objects according to different paradigms. Two examples called MVC and DCA are discussed in depth in later sections. The notion of a BabyComponent is recursive; its member 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 system simplicity.

Aspect oriented programming is a technology for capturing cross cutting concerns in code that spans several classes. In section 3F, I speculate if similar techniques can be used to write code for roles so that the code spans all classes implementing these roles. Finally, in section 3G, I show that packages are not applicable to the clustering of run time objects.

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 with a programming environment that supports an extensible set of object structuring paradigms. Section 4 and section 5 describe my old MVC and my new DCA programming paradigms together with a demonstration implementation in Java [1], [2] . Both paradigms answer the essential 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. Both paradigms exemplify the kinds of object structures I envisage for BabyUML. Both paradigms demonstrate a balance between classes and roles in the code.

Section 4 describes *MVC*, my old Model-View-Controller paradigm [10] 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 *Model*.

---

[1] Java is a trademark of Sun Microsystems, Inc. in the United States and other countries.

[2] The program is given in full on the enclosed CD.

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

Section 5 describes *DCA*, my new Data-Collaboration-Algorithm paradigm. The essence of object orientation is that objects collaborate to realize certain functionality. Many object oriented designs distribute the specification of the collaborations as fragmentary information among the domain objects. In the DCA paradigm, the collaborating objects are explicitly identified by the role they play in an interaction, and the interaction pattern is explicitly defined in terms of these roles as follows:

- The *D* for *Data* part is a simple "micro database" that manages the domain objects.
- The *C* for *Collaboration* part is an object that defines the roles that objects play in an ensemble of interacting objects. The collaboration also binds the roles to objects by executing queries on the set of Data objects.
- The *A for Algorithm* part is a method that specifies an interaction. The method is expressed in terms of the roles objects play in the interaction; the binding from role to object is done in the collaboration.

The MVC/DCA experiment reported in sections 4 and 5 is completed. It has revealed the kind of high-level structures that shall be supported by the BabyUML discipline of programming.

The next major step in the BabyUML project is to create *BabyIDE*, an integrated development environment for BabyUML. The experiment will be done in a *BabyIDE laboratory* where I will try out novel semantics for classes and metaclasses together with tools for design, compilation, and inspection.

Section 6 describes a rudimentary BabyIDE laboratory together with its core concepts. The laboratory is embedded within a Smalltalk stored program object computer. Its main feature is that it gives the systems programmer full control over the semantics of classes and metaclasses. Its foundation is a deep understanding of the implementation of objects, classes, instantiation and inheritance.

The laboratory will initially be used 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 to define their collaboration. 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.

The BabyUML project will be completed when it has produced a BabyIDE working prototype that can act as a specification for a commercial, generally applicable software engineering product. Most products will not need the flexibility of a laboratory and can be written in any language.

2 AN EXAMPLE AND A PROBLEM

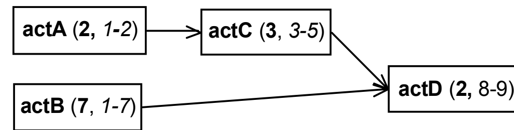
2A An Activity Network Planning Example

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.

Some of the activity attributes are *name*, *duration*, a set of *predecessor* activities, a set of *successor* activities, *earlyStart* time, and *earlyFinish* time. Predecessors and successors are called *technological dependencies*. The *earlyStart* of an activity is when all its predecessors are finished. The *earlyFinish* is most simply computed as *earlyStart* + *duration*. There are more sophisticated forms of technological dependencies. For example, it is possible to start the painting of one wall before the paneling of all walls is finished. Such cases are catered for with various kinds of *activity overlap*.

Frontloading is the calculation of the *earlyStart* and *earlyFinish* times of each activity given the earlyFinish times for their predecessors. 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.

Fig. 1: The experimental activity network



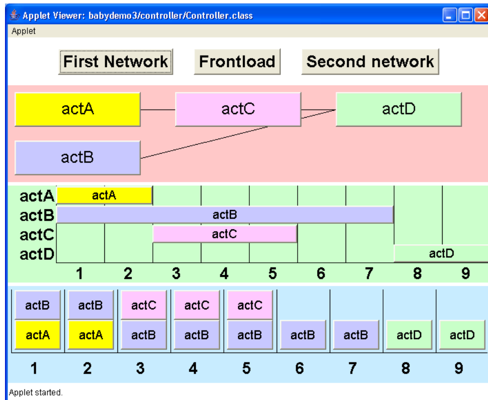
Activities may be tied to *resources*. The creation of the design of a house requires some hours of work by an architect and a draftsman. The digging of the pit requires machinery and the efforts of some workers. *Resource allocation* is to reserve resources for each activity. It is a non-trivial operation; one can easily end up with unimportant activities blocking the progress of critical ones. (We cannot dig the pit because the workers are busy leveling the garden.) There is a single resource in this illustrative network example, say a pool of workers. The resource has unlimited capacity and an activity employs a single worker for its duration.

The example has been programmed in Java as an illustration of the concepts discussed in this chapter[1]. The user interface (GUI) is shown in figure 2. It is partitioned into four strips. The top strip has three command buttons: *First Network* that creates the network shown in figure 1. *Frontload* the network and allocate resources. *Second Network* that creates another network in order to demonstrate 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

[1]The complete Java code can be found on the enclosed CD.

loading along the vertical. The snapshot in figure 1 has been taken when *actA* has been selected.

Fig. 2: The Java program user interface



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

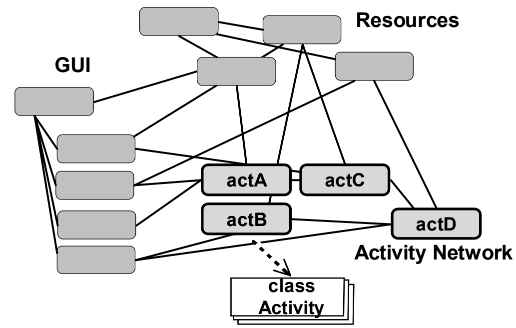
2B My old style doesn't always scale

A potential problem with my usual programming style is easily demonstrated. Figure 3 illustrates how I would normally implement the network example. The rounded rectangles denote objects, the solid lines denote links between them, the white rectangles denote classes, and the dashed 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. The technicalities of the user interface have been separated from the domain objects in conformance with the MVC paradigm; the View and Controller objects are shown on the left.

My usual implementation style tends to give fairly small objects in a distributed structure and with distributed control. This leads to a large number of links and 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. This works fine in simple cases, but it can degenerate into a bowl of spaghetti for very large systems.

Fig. 3: A typical application



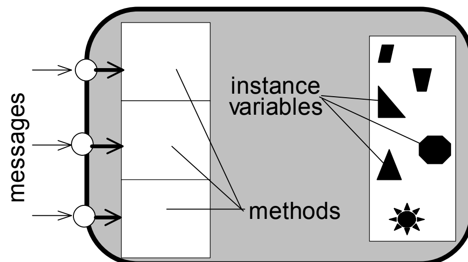
Every object is an instance of a class written in a language such as Simula, Java, or Smalltalk. The structure and domain logic is distributed among the methods of the classes with their superclasses. This fragmentation makes it hard to see the system as a whole. Any spaghetti that may be in the design will effectively be chopped into noodles in the classes. The structure is in the mind of the beholder and not explicit in the code; so my beauty can be your noodles.

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. [11] The concepts were considerably refined in the Smalltalk language and run-time system. [9]

Fig. 4: The object



Objects are entities that encapsulate state and behavior. In this chapter, I 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 method for handling the message. 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 different ways.

In some contexts, an object is defined as an instance of a class. A more conceptual definition is preferred in BabyUML: *An object is an entity that encapsulates state and behavior.* This allows me to focus on the objects and work with different abstractions for different purposes. The class abstraction discussed in section 3B describes the nature of a set of objects. The role abstraction discussed in section 3C describes an object's contribution made by a set of objects in a structure of collaborating objects.

The concept of an object is specialized in the *BabyComponent* that is introduced in Section 3D.

3B The class

In most object oriented languages, an object is an instance 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 object because it is the object that holds the state and the methods are executed in the context of the object.

A class inherits all the features of its superclasses, it can add features of its own, and it can override methods defined in the superclasses. A class with its superclasses can always be flattened into a single class with no superclass. This means that

[1]This in contrast to a procedure call that uniquely identifies the procedure body. Also in contrast to Java where the instance variables are visible from outside the object.

the actual distribution of features between the superclasses does not in any way influence the semantics of the object, and I see class inheritance mainly as a very powerful device for code sharing.

The class concept is important in BabyUML, but its use is restricted to describing isolated objects. The state and behavior of ensembles of collaborating objects are described by the role models of the next section.

3C The Role Model

Prokon was to be a comprehensive system for planning and control [12] that we worked on in the early seventies. The system architecture was based on objects negotiating on behalf of the line managers and depended on global control of the object interaction patterns. The line managers should own their objects with the corresponding classes. Objects playing the same roles in the interactions could, therefore, be implemented by different classes owned by different managers. We tried implementing the system in Simula [11], but failed because the Simula language insisted on our knowing the class of every object. Indeed, there was no notion of an object with an unknown class.

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 the Smalltalk dynamic typing let me focus on object interaction independently of classes and class hierarchy. The MVC paradigm discussed in section 4 was a result of thinking in terms of objects rather than classes, but there was still no construct for explicitly programming the interaction patterns.

The experience with MVC led me to search for a new abstraction that let me work explicitly with the interactions. The result was *role modeling*, an abstraction that describes how an ensemble of objects interact to accomplish some desired result. Each object has a specific responsibility in an interaction[1]; we say that it plays a specific *role*.

We developed role modeling tools for our own use in the early eighties. Our tools were demonstrated in the Tektronix booth at the first OOPSLA in 1986. The first mention in print was in an overview article by Rebecca Wirfs-Brock and Ralph Johnson. [14] Our own report was in an article in JOOP in 1992. [15] My book, *Working with Objects* [6], explains role modeling in depth. A theory of role modeling is given in Egil P. Andersen's doctoral thesis. [16]

Some of the role modeling concepts have made it into the UML Collaborations and Interactions packages as follows (my emphasis):

> ***Collaborations*[2]**
>
> *Objects in a system typically cooperate with each other to produce the behavior of a system. The behavior is the functionality that the system is required to implement.*
>
> *A behavior of a collaboration will eventually be exhibited by a set of cooperating instances (specified by classifiers) that communicate with each other by sending signals or invoking operations. However, to understand the mechanisms used in a design, it may be important to describe only those aspects of these classifiers and their interactions that are involved in accomplishing a task or a related set of tasks, projected from these classifiers.* **Collaborations *allow us to describe only the relevant aspects of the cooperation of a set of instances by identifying the specific roles that the instances will play.*** *Interfaces allow the*

[1] More about responsibility driven design and roles in [13].

[2] Extract from section 9.1 in OMG document formal/2007-02-03. Reprinted with permission. Object Management Group, Inc. (C) OMG. 2007.

externally observable properties of an instance to be specified without determining the classifier that will eventually be used to specify this instance. Consequentially, the roles in a collaboration will often be typed by interfaces and will then prescribe properties that the participating instances must exhibit, but will not determine what class will specify the participating instances. [7]

A role model is analogous to a stage production. *Hamlet* is a tragedy written by William Shakespeare. In a certain production; the role of Hamlet may be played by the actor Ian, Ophelia by the actress Susan. Outside the stage, Ian and Susan live their regular lives. Other productions of the same play may cast different actors. Role modeling sees a system of interacting objects as a stage performance:

- A set of objects is like a set of available actors.
- An object interaction is like a stage performance and objects play roles just as actors do.
- A role model corresponds to a drama. Both describe what shall take place in terms of roles and their actions. Neither specifies objects, classes or specific actors.
- *A BabyUML discovery is that the selection and assignment of objects to roles can be done by a query on the objects just as the selection and assignment of actors to roles is the task of casting.*
- A role may be seen as an indirect link to one or more objects.
- A role really exists only while it is being played, i.e., when it is bound to one or more objects. At other times, there may be no object or actor assigned to the role. Therefore, the role concept is a dynamic concept.

As a role model example, we will consider the *Observer Pattern* as described in the *Design Patterns* book. [17] 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 a textual description, a class diagram, and a kind of collaboration diagram. I will here describe it with a role model.

The essence of object orientation is that objects collaborate to achieve some desired objective. 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 structure of roles that work together to reach the objective and the links between these roles.

Fig. 5: The Observer role model

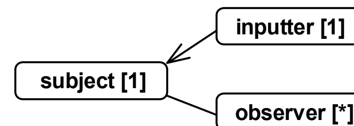
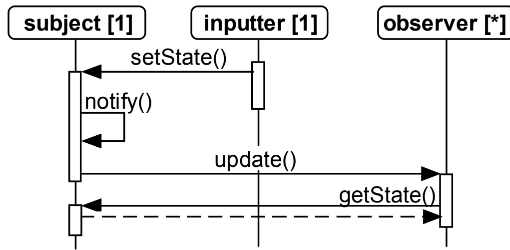


Figure 5 show the Observer pattern as a role model. We see three roles. There is one object playing the subject role. There is one object playing the inputter role. There is any number of objects playing the observer role. They are all linked to the single subject, and the subject is linked to them all.

Every object has a unique identity. A role name such as subject is an alias for one or more objects, it can be seen as indirect addressing with dynamic binding between role and objects. We use a role name as an abbreviation of: *"the object or objects that play this role at a certain time and in a certain context"*.

Figure 6 specifies how the objects interact when synchronizing subject and observer. We see that inputter sends setState () to subject, presumably changing its state. subject then sends an update () message to all observers. The observers finally interrogate the subject to get the new state.

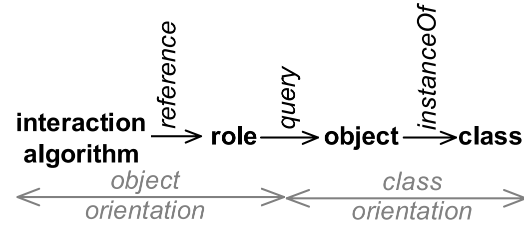
Fig. 6: An Observer interaction[1]



Note that many objects may play the observer role in different contexts and at different times, but we are only concerned with the objects that play the observer role *in an occurrence of the interaction*. Also note that a role may be played by many objects and an object may play many roles. In this example, an object playing the inputter role could also play the observer role. The collaboration diagram in *Design Patterns [17]* mandated this by showing two objects called aConcreteObserver and anotherConcreteObserver respectively; the first also playing the inputter role.

A role modeling tool called *OOram* was put on the market, but the interest was not sufficient to sustain it as a product.

Fig. 7: Bridge between roles and classes



The reason for the OOram failure could be that I had not found a conceptual bridge between roles and classes. I have recently found this bridge; *a role is bound to a set of objects through a query*. The relations are illustrated informally in figure 7. Object interaction is specified by an algorithm. The algorithm references the interacting objects indirectly through their roles. A query binds a role to one or more objects. An object is an instance of a class. There is no restriction on the formulation of a query. Its results may vary over time so that the role is a dynamic notion. The nature of an object does not change over time so that the class is a static notion.

An implementation of this unification is described in section 5 on the DCA paradigm.

3D The BabyComponent

Section 2B demonstrated my need for injecting some sort of object clustering into my systems. The instance variables in the objects are in themselves less than useful for this purpose. 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 may 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 object

[1]This BabyUML sequence diagram describes sequential interaction. A filled arrow is a method call. A thin, vertical rectangle denotes a method execution. The objects bound to the observer[*] role work in lock-step; their updates appear to occur simultaneously.

structure from the perspective of a single object, we do not see the structure as a whole.

The UML definition of *Composite Structures* provides the idea. In [7], we find the following[1]:

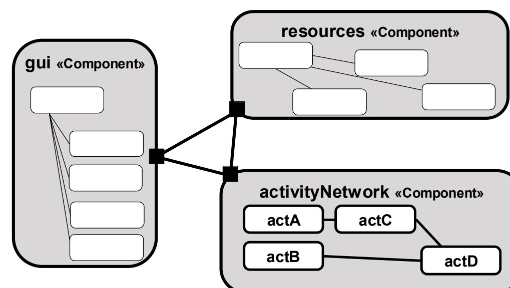
> ***9.1 Overview***
>
> *The term "structure" in this chapter refers to a composition of interconnected elements, representing run-time instances collaborating over communications links to achieve some common objectives.*
>
> ***Internal Structure***
>
> *The InternalStructure subpackage provides mechanisms for specifying structures of interconnected elements that are created within an instance of a containing classifier. A structure of this type represents a decomposition of that classifier and is referred to as its "internal structure."*

A *babyComponent* is an object that encapsulates other objects and can loosely be described as an instance of a UML Composite Structure. A *babyComponent* 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 interconnected Member Objects.

Figure 8 illustrates how the spaghetti of figure 3 can be replaced by a simple structure of three interacting components. The notion of a BabyComponent 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.

[1] Extract from section 9.1 in OMG document formal/2007-02-03. Reprinted with permission. Object Management Group, Inc. (C) OMG. 2007.

Fig. 8: 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. My code can specify how components are interconnected and how they interact. The code can thus document 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 firewalls for security and privacy. Indeed, it is hard to see how privacy and security can be achieved without some form of enforced component architecture.

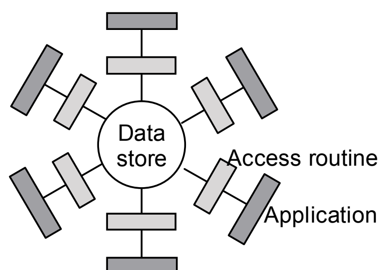
The notion of a BabyComponent is useful in many contexts. A specialization is the DCA component described in section 5.

3E The Database

An early idea for system structuring was the idea of separating system state and system behavior. From the first, 1963 version, our Autokon CAD/CAM ship design system [24] 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 application

programs 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 9.

Fig. 9: Separating data and procedure



This separation of state and behavior is very useful for our purposes. Consider the roles and classes illustration in figure 7. Put the objects of figure 7 in the data store and you get the Data of the DCA paradigm. Put the role definitions with their queries into the access routines and you get the Collaboration of the DCA paradigm. Put the interaction methods into the applications and you get the Algorithms of the DCA paradigm. The DCA paradigm is discussed further in section 5.

3F 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, AOP, [23] 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 roles and interactions can be such aspects since they cut across class boundaries. A technology similar to AOP should be able to support methods that are defined for a particular role and thus shared among all classes that implement this role. These classes may specialize the role methods as needed. 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? An interesting thought for a future experiment.

3G The Package

A UML *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.

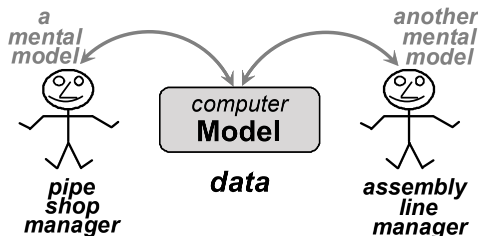
An object is an instance of a class. The classes in its superclass chain are typically members of different packages. An object is thus related to several packages. The notion of a package relates to compile-time issues and is irrelevant in the context of interacting, run-time objects.

4 MVC: THE MODEL-VIEW-CONTROLLER PARADIGM

How can we build a system that is experienced as an extension of the user's 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 it presents an image of the world that corresponds to the user's 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 model and the information represented in the computer. The idea is illustrated in figure 10.

Fig. 10: 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 would be specialized according to the needs of their owners, yet they could all interact according to a common scheme.

I implemented the first MVC while being a visiting scientist with the Smalltalk group at Xerox PARC. [10] 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 it was too limiting to focus on one object at the time. The second 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. This would violate another Smalltalk ideal; namely that code should be visible, simple, and lucid.

4A The MVC 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 [19]:

> *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[1].

So the user's mental model is *information*, information does not exist outside the human brain. But *representation of information* can and do exist outside the brain. It is called *data*. In the network 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 model of the project.

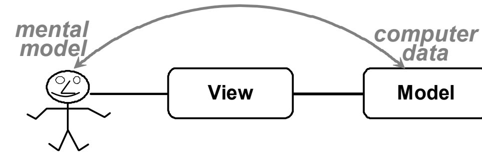
I will discuss the Java implementation of the View and Controller below, and the Model with its links to the View-Controller pair in section 5.

4B The MVC View

The *View* transforms the latent Model data into a form that the human can observe and convert into *information* as illustrated in figure 11.

[1] ©IFIP 1966. Quoted with permission.

Fig. 11: The View couples model data to the information in the user's brain so that they appear fused into one

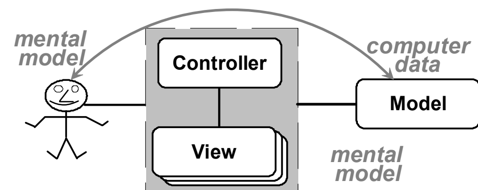


I will discuss the Java implementation in section 5E.

4C The MVC 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.[2] [3]

Fig. 12: The Controller creates and coordinates multiple Views



Looking back to section 3C on role models, we realize that *Model*, *View*, and *Controller* are *roles*

[2] Note that a Smalltalk 80 Controller is only responsible for the input to a single view. It is thus different from the one discussed here, see [18].

[3] Also note that some so-called MVC structures let the controller control the user interaction and thus, the user. This idea is fundamentally different from MVC as described here. I want the user to be in control and the system to appear as an extension of the user's mind.

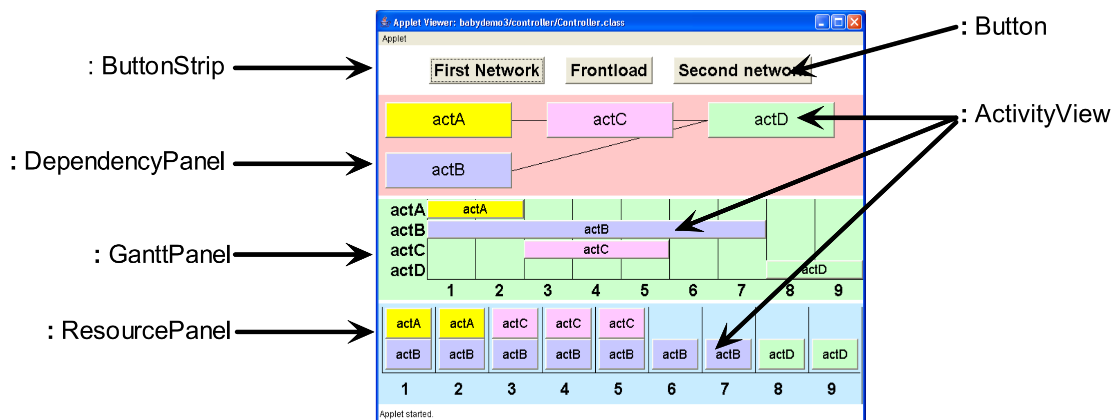
played by objects. Their classes are unspecified and irrelevant to the MVC paradigm.

4D The anatomy of the Java user interface code

The GUI for the network example was shown in figure 2. Figure 13 shows the same GUI annotated with the implementation class names for its main parts. We see that the four strips of the tool are

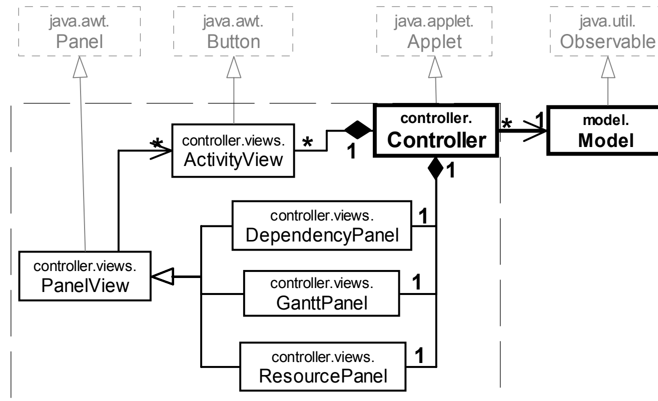
1. The top strip is an instance of class ButtonStrip; it contains command buttons.
2. The second strip is an instance of class DependencyPanel; it is a view that shows the activities with their technological dependencies.
3. The third strip is an instance of class GanttPanel; it is a bar chart showing 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. 13: 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 14. 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 14. 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.

Fig. 14: Java class diagram



I will go into more details when I discuss the Model internals and system behavior in section 5.

4E Controller code coordinates selection

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

Fig. 15: actA is selected in all views where it appears

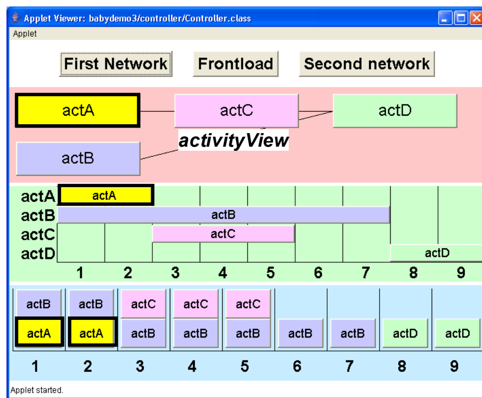
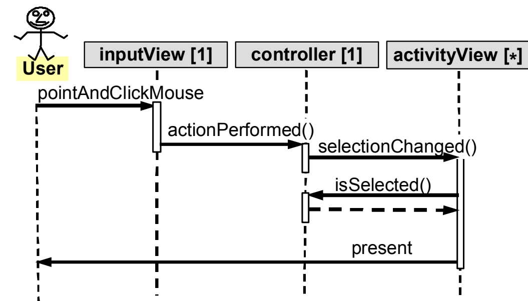


Figure 15 shows the tool after the user has clicked on any of the actA activity views. The key to simplicity and generality is that the view being clicked only reports this event to the controller object. The controller decides that this is indeed a selection command, and that it shall be reflected in the appearance of all activityViews, including the one that was clicked. This behavior is illustrated in the BabyUML sequence diagram of figure 16.

Fig. 16: The selection interaction



In this program, the inputView role happens to be played by an instance of class ActivityView. We see from figure 14 that ActivityView is an awt.Button, so it sends an actionPerfomed event to its actionListener. All activityViews are created to let the controller be their actionListener.

Discussion 1. A variant of the selection interaction could use the Observer pattern[1] 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 is the simplest and does not restrict flexibility and extensibility.

Discussion 2. We see from figure 14 that the controller knows both panels and activityViews. An alternative could be to 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 not need to know the inner workings of the panels. I did not choose this solution because I wanted to keep the experimental program as simple as possible.

[1]See *section 3C.*

5 DCA:
THE DATA-COLLABORATION-ALGORITHM PARADIGM

I now come to the Model part of MVC. Seen from the Controller, it looks like an ordinary object. But a single object that represents all activities and resources would be a monster. My new *DCA paradigm* tells me how to master a monster object by clearly separating roles from objects and by creating bridges between them as illustrated in figure 7.

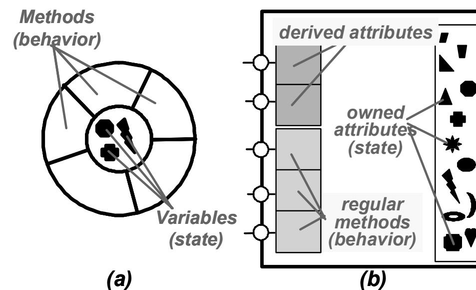
The Model of the MVC paradigm is implemented as a *DCAComponent*. It looks like a regular object from the outside, characterized by its provided operations. Inside, there is a well ordered and powerful object structure partitioned into three parts, *Data*, *Collaborations*, and *Algorithms*. I hinted at the nature of these parts in the introduction and will now go into the details.

5A The MVC Model part as a single object

The Java tutorial [20] describes an object as a number of variables (state) surrounded by methods (behavior) as illustrated in figure 17(a). This is actually a better illustration of the Smalltalk object than the Java object. In Smalltalk, the variables are invisible from outside the object; all access has to be through the methods.[1]

Figure 17(b) shows an alternative illustration that I use as a starting point for discussing DCA. Borrowing terminology from UML, I use the term *owned attributes* to denote the object state (fields, instance variables). I use the UML term *derived attributes* to denote attributes that are computed rather then 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. 17: The object as an instance of a class.
a) The object as depicted in the Java tutorial.
(b) A more accurate object model



Behavior is activated in an object when it receives a message. The message is dynamically linked to the appropriate method, and the method is activated. This link is symbolized by a small circle on the object boundary in figure 17(b).

[1] 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.

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

Fig. 18: The DCA component

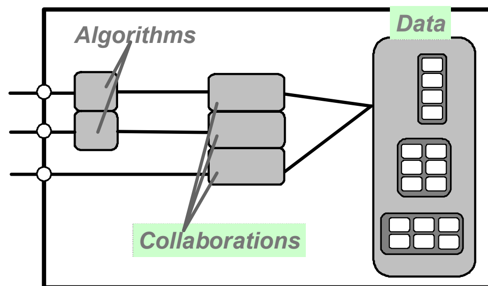


Figure 17 shows an object as an entity that encapsulates state and behavior. Figure 18 illustrates the *DCA component*. It looks like the object of figure 17 when seen from its environment. Inside, there are a number of specialized parts: Data, Collaborations, and Algorithms.

5B1 The D stands for Data

The Data part corresponds to the variables (owned attributes) of the regular object. The variables 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. The schema can be read and understood independently of the system around it; an important step towards system simplicity:

- The domain objects are organized in a number of relations in the first normal form, ensuring referential integrity.
- The structure is represented in explicit relations. Contrast with my traditional representation where structure information is fragmented 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 merely implement some Java classes in the network example.

I do not assume persistence, concurrency, access control, security, or any other goodie usually associated with databases. There is also an addition to mainstream database technology; the DCA data are encapsulated within a component so that the system as a whole can include a hierarchy of independent "micro databases".

5B2 The C stands for Collaboration

DCA *Collaborations* correspond to the derived attributes of the regular object. Algorithms access the domain objects through the roles these objects play. Collaborations bind roles to domain objects through queries as illustrated in figure 7. 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 algorithm code. The notion of Collaborations is derived from the OOram role model [6] and corresponds to the external views used in database technology.

In this experiment, collaborations are coded as classes that have the collaboration roles as attributes and the database queries as methods. A *DCA Collaboration* is an instance of such a class where the results of the queries are assigned to the role variables, thus binding roles to actual domain objects. A binding is only valid in a certain context and at a certain time and realizes a kind of dynamic, indirect addressing, [1]

[1] The DCA Collaboration corresponds to the UML *CollaborationUse*. My choice of name reflects my focus on objects rather than classes.

Objects using a Collaboration see the Data in a perspective optimized for their needs. Note that these user objects can be internal or external to the Model.

5B3 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 are properties of the inter-object space. The interaction algorithms are coded in separate classes in BabyUML, 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 the system dynamics.

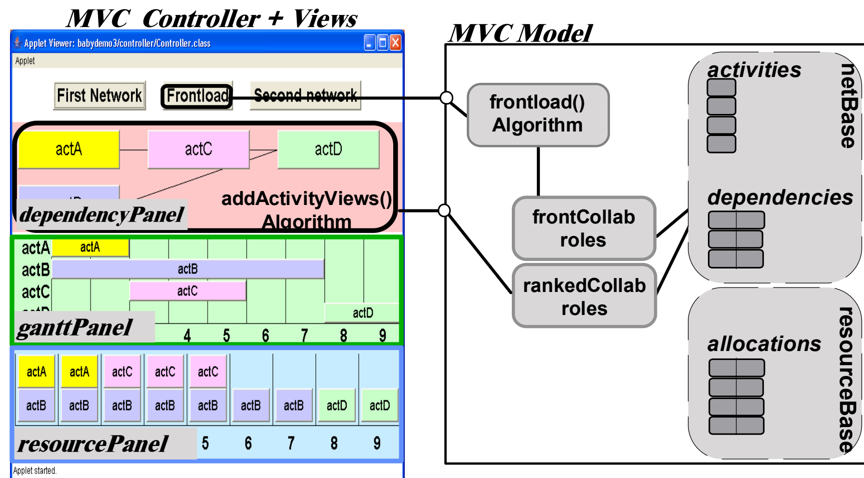
5C The MVC Model part as a DCA component

The Model part of the network example is implemented as a DCA component. Some important objects are shown in figure 19. For illustrative purposes, I have separated the Data into two sub-parts. The netBase holds the activity network in two relations. The activities relation is a list of Activity objects. The dependencies relation is a list of Dependency objects, each having a predecessor and successor attribute. The resourceBase has a single relation, *allocations*, that is a list of Allocation objects, each having a time and an Activity attribute.

We have previously seen that 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 Algorithm. The Algorithms are users of the DCA Data and access them through suitable Collaborations.

In the following, I will discuss the code for the dependencyPanel and frontload buttons together with their algorithms and data access collaborations as illustrated in figure 19.

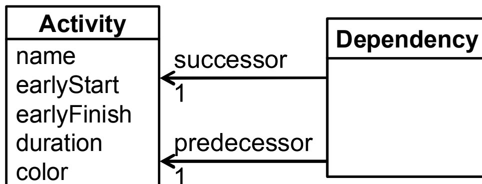
Fig. 19: The MVC Model part as a DCA model.



5D The Data structure defined by a schema

The Data parts are defined by their schemas. Figure 20 shows the netBase schema expresses as a UML class diagram.

Fig. 20: The *netBase* schema as a UML class diagram

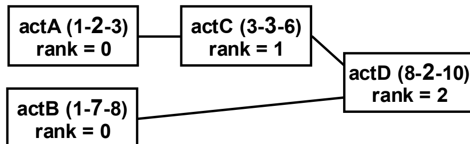


The corresponding Java class declarations are trivial.

5E Example 1: Panel layout

Figure 21 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.

Fig. 21: The ranked activities



The DependencyPanel layout Algorithm is as simple as can be. (Too simple, actually, it will often lead to overlapping dependency lines.) The most interesting statements in the DependencyPanel class are as follows:

```
DependencyPanel::private void addActivityViews() {
....
for (int rank=0; rank <= rankedCollab.maxRank(); rank++) {
  ....
   for (Activity act : rankedCollab.activityListAtRank(rank)) {
     ActivityView actView = new ActivityView(controller, act, 24) ;
     ....
     add(actView);
     ....
   }
 }
}
```

This layout algorithm accesses the activity objects 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 the fifth line of the above code.

The rankedCollab object has a simple recursive method for computing the rank of all activity objects. The rankedCollab object could safely cash the results because it is an observer of the network as a whole and can recompute the rank when necessary.

```
   RankedCollab::public List<Activity> activityListAtRank(Integer rank)
{
    List<Activity> activityListAtRank = new ArrayList<Activity>();
    for (Activity act : netBase. activities()) {
       if (rankOf(act) == rank) {
          activityListAtRank.add(act);
       }
    }
    // Hack. Sort to ensure always same diagram.
    Collections.sort(activityListAtRank, NAME_ORDER);
    return activityListAtRank;
 }
```

5F Example 2: Frontloading

Frontloading is the calculation of the earlyStart and earlyFinish for each activity given the start time of its predecessors. We see from figure 21 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 earlyStart and earlyFinish for actC. actD can finally 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. A default method could look like the following:

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

The problem with this simple solution is that the method cannot be triggered in an activity object before the earlyFinish of all predecessors are known. This means that the frontload network operation belongs in the inter-activity space and should 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 would be overloading a class that should be clean and simple. So the top level frontload () method is coded in a separate FrontloadAlgorithm class. Three problems need to be resolved:

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.

I will discuss the coding of these actions in turn.

5F1 FrontloadCollab, the frontloading collaboration

I have chosen a query-based solution to illustrate how a query result changes through the frontloading process. An activity is *ready to be loaded* if it is not yet loaded and if all its predecessors have been loaded. Here is the query that finds a candidate activity for frontloading expressed in an unspecified language:

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

The FrontloadCollab code is not trivial, but it is nicely isolated giving an attractive separation of concern. The complete code for class FrontloadCollab can be found on the enclosed CD.

5F2 FrontloadAlgorithm, the frontloading interaction algorithm

The frontload algorithm can be expressed in terms of the frontloader role and can loop until frontCollab fails to bind that role to an object. I can give the code to a colleague and ask her to audit and sign it. The frontloading interaction is implemented in the FrontloadAlgorithm class:

```
FrontloadAlgorithm::public void frontload (Integer startWeek) {
.....
  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);
  }
}
```

We see that frontCollab defines two roles; frontloader is the activity object being loaded, and

frontPredecessors are its predecessor objects. The frontload code is pure algorithm with no confusing side issues. It is thus a good starting point for dealing with more complex situations.

5F3 The frontloading earlyFinish algorithm in the Activity class

The activity object is responsible for all activity properties and can compute its earlyFinish when the earlyStart is known:

```
public class Activity {
private Integer earlyStart, earlyFinish, duration;
...
public void setEarlyStart(Integer week) {
  earlyStart = week;
  earlyFinish = earlyStart + duration - 1;
}
...
```

6 THE BabyUML LABORATORY

I am now entering upon a new stage in the BabyUML project and find it opportunely to restate the project goal:

> *The goal of the BabyUML project is to increase my confidence in my programs. The keywords are simplicity and leverage. Simplicity helps me to think clearly and a reader to understand and audit my code. Leverage lets me say more with less. The end result shall be a new interactive development environment with appropriate languages and tools for supporting high level abstractions.*

The results reported in the previous sections have revealed the kind of abstractions that shall be parts of *BabyIDE*, the BabyUML integrated development environment

My completed experiments have clarified the notions of a *class* to describe the nature of an object and the notion of a *role* to describe its place in an interaction. I also have the notion of a *collaboration* to select the objects that play certain roles at certain times and in certain contexts. It is now time to turn to the tool. What does it take to create a tool that implements roles and classes on an equal level? The notion of a class is well covered in current programming languages, but the notion of a role is more elusive. I need to lift the role to the same level as the class; I already know how to bridge the gap between them.

The next stage is to experiment and try out novel semantics for classes and roles together with tools for design, compilation, debugging, and inspection.

In the Java network example, the roles were represented as attributes in collaboration classes and the queries were coded as regular methods in those classes. The notion of a role was in my head, not in the code. Another weakness was that the language for defining the DCA Data schema was regular Java where I would have preferred a declarative language. I need to experiment with different notions of classes, roles, components, and other concepts in my next move towards the BabyUML goal. I need a *BabyUML laboratory*.

The nature of an object is specified by the object's class; the object is an *instance of* that class. Different kinds of objects are instances of different classes. But the concept of a class is rigidly defined in the language specification of common object oriented languages such as Java. Contrast this with the Smalltalk stored program environment. Classes are here represented as regular objects. This means that the concept of a Smalltalk class is defined by *its* class, a metaclass. Classes and metaclasses come in pairs, the metaclass being responsible for static methods and variables. The notions of classes and metaclasses are defined in the default Smalltalk class library. I can complement them with my own versions by implementing my own ideas. The extreme flexibility of Smalltalk makes it an ideal foundation for my BabyIDE laboratory. This does not mean that a future BabyIDE product need be written in Smalltalk; the product can be written in any language based on a specification expressed as a Smalltalk prototype.

BabyIDE starts from a simpler basis where an object is an instance of a class, a class is an instance of *MetaSimpleclass*, this metaclass is an instance of *MetaMetaclass*, and *MetaMetaclass* is an instance of itself. Other kinds of classes will later be added by adding new metaclasses. The *instantiation hierarchy* is essential for understanding the nature of all objects.

A class may be subclass of another class that may be subclass of some other class, etc. This *class inheritance hierarchy* is useful for code reuse and, to a certain extent, for organizing the programmer's thoughts. The nature of an object is not influenced by the actual ordering of classes in a chain of superclasses because the chain can always be refactored into a single class as described in section 3B.

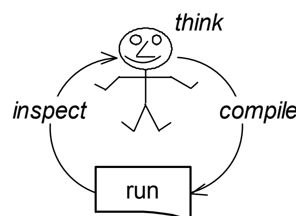
The instantiation and inheritance hierarchies are orthogonal. The human brain does not seem well equipped to deal with two hierarchies simultaneously, and the instantiation and inheritance hierarchies have been confused by better brains than mine. Yet progress in the BabyUML project depends upon alternative definitions of the concepts of class and metaclass. As a start, I have implemented the core objects of a BabyUML laboratory including my own versions of class and metaclass as a foundation for further experiments.

Section 6A describes what I mean by an integrated development environment as distinct from a modeling tool equipped with code generators. Section 6B introduces the BabyUML object notation. This is a notation for the run-time objects as distinct from the well known notation for compile-time classes as they appear in UML class diagrams. Section 6C describes the BabyUML laboratory as it is embedded within the Smalltalk stored program object computer. This section is somewhat detailed because I find that its careful attention to run-time objects is an excellent antidote to the potential confusion of instantiation and inheritance.

6A The Integrated Development Environment

Nusse, the first Norwegian computer, was deployed in 1953. Its smallest addressable unit was a *word* of 32 bits. When I started programming in 1958, I met a computer where 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, and 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 22 illustrates the situation. The man-machine system was harmonious because the same conceptual framework applied throughout my thinking, coding, debugging, and inspecting.

Fig. 22: The programmer's mind and the computer



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 22 was again closed when the plain compilers grew into *integrated development environments*. First for FORTRAN, today I use the Java NetBeans IDE[1]. I think, code, inspect, debug,

[1] http://www.netbeans.org/

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 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 for my simple problem 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 have to inspect and debug in terms of Java. The correspondence with my MVC and DCA models is far from simple. Harmony is lost.

6B The BabyIDE Object Notation

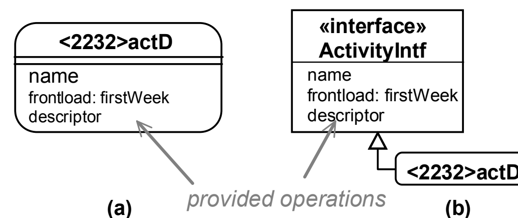
The BabyIDE is centered on objects and their interaction. Interacting objects can only see the *provided operations* of their collaborators. Conceptually, an object appears to encapsulate state and behavior. BabyIDE introduces new notations for objects. A notation for the external properties of an object is described in section 6B1. A notation for the object as a conceptual entity is described in section 6B2. The object implementation with class and superclasses is discussed in section 6C.

The notations presented here symbolize concrete objects with their identity, state, and behavior. The notations will later be modified to symbolize roles.

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 23 to denote an object seen as a black box.

Fig. 23: Examples of the encapsulated projection



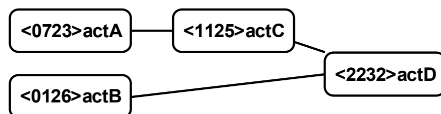
The BabyIDE notation for an object is a rounded rectangle; its corners are rounded to distinguish it from the UML classifier. An object has a unique identifier, the objectID that is shown in angle brackets <...>. Some objects have a name; this is

then shown after the objectID. There are two equivalent notations. The inline form in figure 23 (a) is useful in simple diagrams. The compact form of figure 23 (b) uses the UML symbol for an interface to show provided operations. A tool can pop up the interface dynamically so as to save screen acreage.

Note that I use the Smalltalk syntax throughout this section. For example, name is equivalent to the Java name() method call.[1] The Smalltalk frontload: firstWeek is equivalent to the Java frontload(firstWeek).

The encapsulated object notation is useful in "wiring diagrams" showing structures of interlinked objects as illustrated in figure 24.

Fig. 24: "Wired" objects implementing the planning network example



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 and superclasses, but we shall 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 all that it encapsulates. The result is the *conceptual object*; very effective as a concept and very inefficient if naïvely implemented. We again use a rounded rectangle to denote an object and get the white box view of figure 25. 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 some or all of the names and values of the object's instance variables.
- The bottom compartment shows some or all of the operations. A tool could also show the code that implements a selected operation so that it can be inspected and edited.

Fig. 25: Example conceptual, white box object projection

<2232> actD	
activityName	**'actD'**
duration	**2**
earlyStart	**8**
earlyFinish	**9**
predecessors	**{<0126>. <1125>.}**
successors	**{ }**
name	
frontload:	
inspect	

6C The Smalltalk Stored Program Object Computer

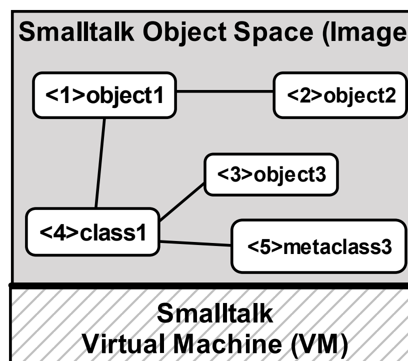
Smalltalk [9] is the ideal proving ground for BabyIDE. The Smalltalk notions of class, method, programming language and programming tools are all realized by objects. A new class is created by sending the message new to its *metaclass*; another object. The code for a method is translated from its text 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 alternative ones.

[1] In Java, name is a reference to the corresponding instance variable. Instance variables are invisible from outside the object in Smalltalk, so name must here necessarily be an operation.

In this section, I give a detailed description of how objects and classes are implemented in the default Smalltalk class library. I need this deep understanding of the default Smalltalk way before I can safely create my own variants of almost any of the Smalltalk library classes for the purposes of my new BabyIDE.

Figure 26 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.

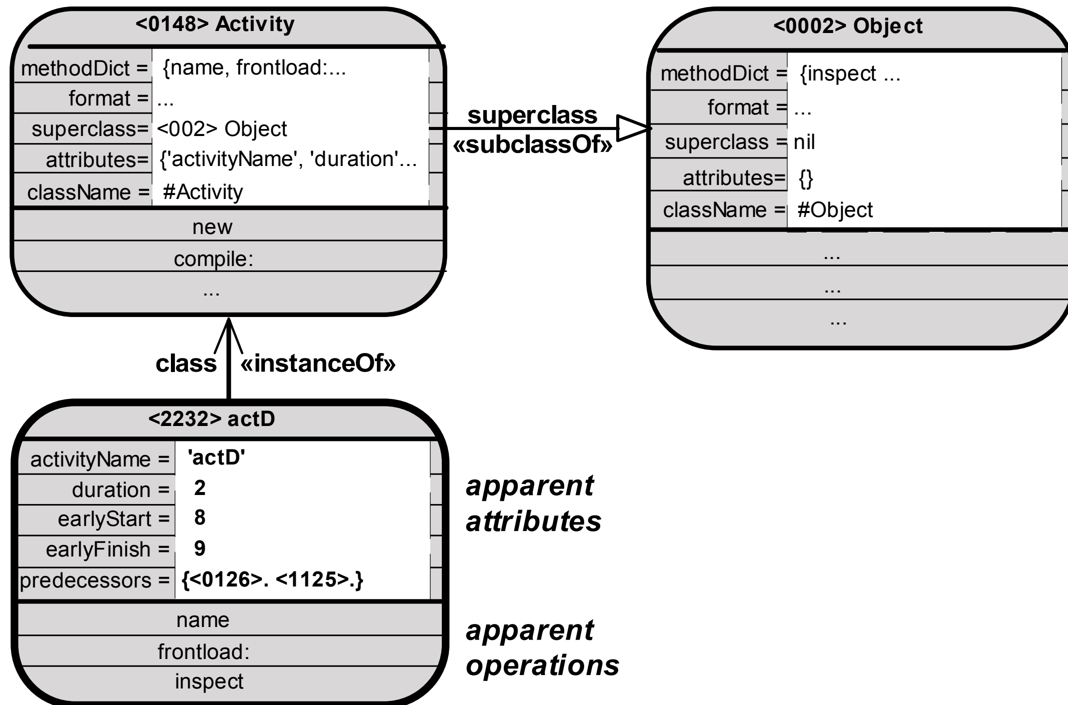
Fig. 26: The Smalltalk stored program virtual object computer



Smalltalk objects are stored in the Object Space, the *image*. The VM creates a new object when told to do so by a class method; returning the objectID of the new object so it can later be the receiver of messages. The VM also removes objects as they become unreachable (garbage collection).

The bytes representing an object are stored on the computer's memory heap. It would be ridiculously inefficient if all the features of an object should be stored in every object. The <2232>actD object only stores its state values as indicated on a white background in figure 27. 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.

Fig. 27: Implementation of the example object.



The objects in figure 27 are as follows:

- The object that is in the center of our interest has objectID=<2232> and stores the values 'actD', 2, 8, 9, {<0126>. <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 attribute of the class and all its superclasses. They are shown on a gray background in the middle portion of the <2232>actD 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 <2232>actD object. A tool can display the corresponding methods.
- The figure shows clearly how 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 VM instructions (byte codes) 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 superclasses. If this search fails, the VM starts the search anew with the default selector doesNotUnderstand: aMessage. 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 (an object) and puts it on the stack (another object). 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 27; 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 27). I look at an object; its features are in the attributes of its class object. I look at the class object; *its* features are in the attributes of the metaclass. This characteristic of "everything is stored somewhere else" is a potential source of extreme confusion. It is here that the BabyUML conceptual object notation proves its worth by showing state and behavior where they belong conceptually, rather than where they happen to be stored.

It is hard to see where the UML class symbol fits in. It does not symbolize the instance, because the instance is a merger 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 the features of the class object.

In my first BabyIDE experiment, I tried to realize a stored program object computer by simply instantiating the UML metaclasses to get my stored program class objects. I failed because there is a fundamental difference between UML class diagram and the corresponding run-time objects. Take the notion of a *link*. In UML, it is defined by three interlinked elements: anAssociationEnd, anAssociation, and anotherAssociationEnd. These three model elements form the high level description of a run-time binary link that can be realized as a pair of instance variables. The model elements must be compiled into the run-time objects; instantiating UML metaclasses to get run-time objects simply does not work. I abandoned this first, naïve approach and accepted that I must clearly distinguish between compile-time and run-time in my experiments.

The conclusion was 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 resolve this difficulty.

The <0148>Activity class object has its own provided operations with the corresponding methods. In the default Smalltalk implementation, it responds to the message compile: sourceCode. The corresponding method is a compiler that translates the sourceCode 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 object has a link to its sourceCode, thus making it feasible to close the loop of figure 22.

6D The BabyUML Laboratory Implementation

I argued for the choice of Smalltalk for implementing BabyIDE in section 6C. The choice of its Squeak dialect [25] was 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 any release 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 that can be applied to BabyIDE are to be found in the Squeak mailing lists and the evolving class and package libraries.

¤ 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.

Fig. 28: The BabyIDE laboratory is embedded in the Smalltalk object space

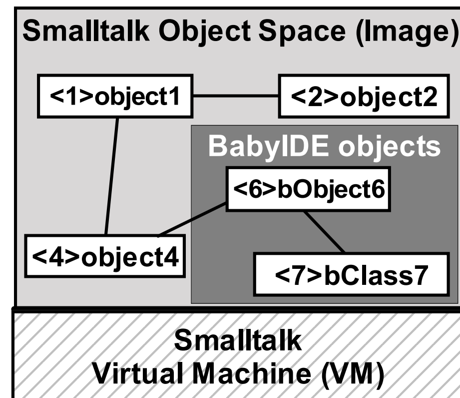
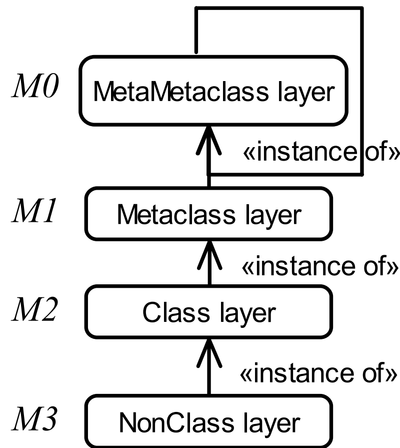


Figure 28 illustrates the BabyIDE implementation. The baby objects have their own classes, metaclasses, and metametaclasses; but they can freely interoperate with regular Smalltalk objects because they all conform to the VM conventions.

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 static attributes and methods. Many Smalltalk novices find it hard to distinguish between regular and static attributes and methods. In BabyIDE, I initially remove the notion of static attributes and methods from the core classes. There is no loss of generality; I can always implement the notions of shared features at a higher abstraction level. The result is a set of clear core constructs that let me explore new languages and tools for my new discipline of programming.

Fig. 29: The BabyIDE Instantiation Architecture



Every object is an instance of a class. 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 depends on circumstances. The core layers from the concrete to the abstract are shown in figure 29:

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

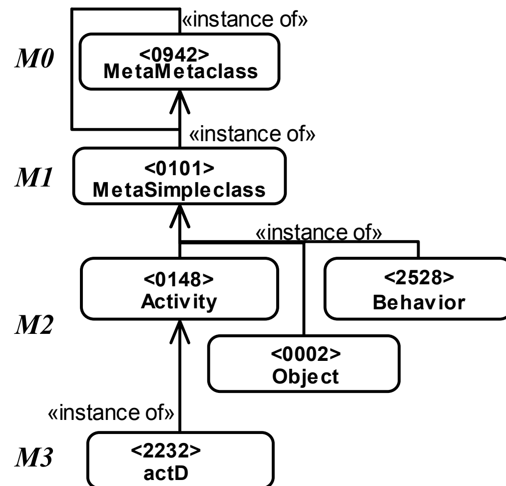
M2 - Class layer: Here are the regular classes. Class objects create new instances, act as repositories for information common to these instances, and know how to translate code from a human form to executable binary.

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 MetaMetaclass. Directly or indirectly, all BabyIDE objects are instances of this class. MetaMetaclass is an instance of itself so it had to be created by a somewhat tricky program.

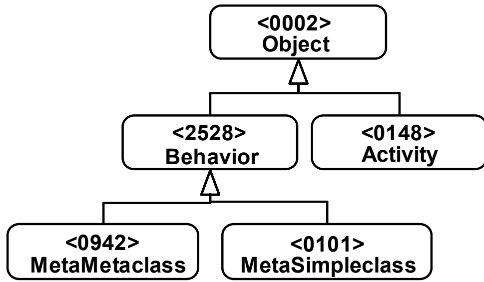
Note that the BabyIDE layered architecture is an *instantiation* hierarchy, the implementation of the <2232>actD object with its class and metaclasses is shown in figure 30.

Fig. 30: The example instantiation structure



The orthogonal *class inheritance hierarchy* is a very powerful device for code reuse and code sharing. Figure 31 shows the inheritance hierarchy of the same network example. This particular solution is not very interesting because the inheritance hierarchy can be refactored without changing the system semantics. We particularly notice that it bears no relationship to the instantiation hierarchy of figure 30.

Fig. 31: The example inheritance structure



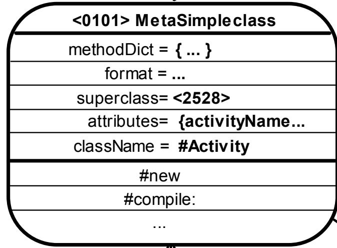
The human mind is well equipped for dealing with a hierarchy, but it finds it harder to handle two of them simultaneously. BabyIDE will gain its power and extensibility by exploiting both the instantiation and the inheritance hierarchies. But this should be isolated to the toolmaker's domain; application programmers should only see a single hierarchy that supports powerful concepts provided by the toolmakers.

6D2 Example implementation of the class layer

Figure 32 shows the actual objects that represent the <u><0148>Activity</u> class. Note that I am using the conceptual object notation for the objects. A diagram showing all the objects would be a complete mess.

Fig. 32: Implementation of the <0148>Activity class

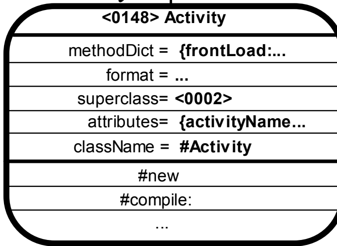
M1: metaclass layer

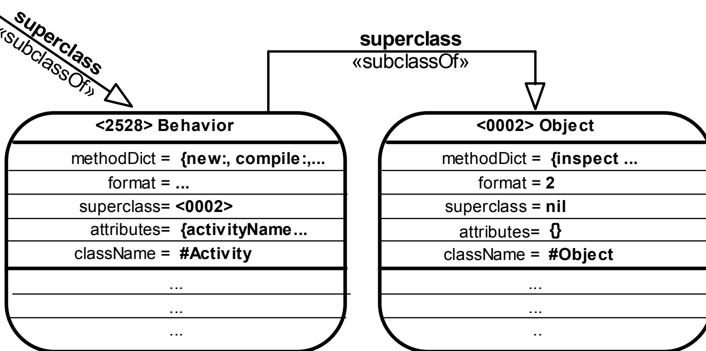


class «instanceOf»

superclass «subclassOf»

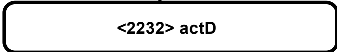
M2: class layer





class «instanceOf»

M3: Nonclass layer



Note that the <0148>Activity class object responds to its own messages such as new and compile:. The corresponding methods are found in the methodDict of <2528>Behavior. Further note that <0101>MetaSimpleclass responds to its own versions of new and compile:. We have to look at the class of <0101>MetaSimpleclass to find the corresponding methods. They may or may not be identical to the <0148>Activity methods.

The theory is simple, but the actual realization gets rather complicated. It is a challenge to device concepts and tools that leverage the potential while hiding the complexity.

7 CONCLUSION

The BabyUML vision is that I shall regain the mastery of my programs by making them *"so simple that there are obviously no deficiencies"*. Early on, it became clear that a solution would have to focus on the inter-object space where we see the objects, the links between them and their interactions. Role modeling provides abstractions for explicitly describing those aspects of object systems, but modeling is not coding. The missing link was the bridge between the roles and the classes.

In a conversation with Kai Fredriksen[1], something he said gave me an aha! *The bridge is a query that finds the object(s) currently playing the role!* This lead to the unification between the role and class abstractions as depicted in figure 7.

I have done a quick Smalltalk implementation of the activity network example of section 2 and spent some time finding a neat way of storing and accessing collaborations with their roles and queries. This led to an interesting observation. An object belongs on the memory heap because it is meaningful until it is garbage collected. A collaboration with its role/object mappings appears to belong on the execution stack because it is only meaningful during an actual interaction. This observation could be the promising start of an interesting investigation.

An early idea was that a database-oriented architecture can be applied to the interior of a composite object. Figure 9 depicts the architecture of the Autokon CAD/CAM system that we deployed at the Norwegian *Stord Yard* in 1963. I believe this was the world's first software product with a database-oriented architecture, and it would be gratifying if the DCA paradigm will prove to be the world's first application of the same architecture in the context of an object.

The MVC has been well known since Jim Althoff and others implemented their own version in the Smalltalk-80 class library. In section 4, I have for the first time described my original MVC idea roughly as it was presented in the original technical notes at Xerox PARC. [10]

The end result of the BabyUML project shall be BabyIDE, a new interactive development environment with appropriate languages and tools for supporting the BabyUML high level abstractions. The role/class unification, the DCA architecture of complex objects, and the MVC as described here form a foundation for further experiments. What remains to be done is to specify, design, and implement BabyIDE. Its top level architecture is compactly expressed in figure 12, where the computer acts as an extension of the user's brain. Seen in this perspective, application programmers are the users of the tools that shall 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 corresponding program descriptions in the computer. This will give added leverage, improved program readability, and reduced program volume. MVC and DCA are but examples; BabyIDE shall be extensible so that it can support many different paradigms, making it an example of Coplien's *multi paradigm design*. [22]

A *first experiment* was to naïvely instantiate the UML metaclasses to get the run-time objects in my stored program object computer. This experiment failed as described in section 6C.

[1] See [21].

A *second experiment* was to implement my own versions of class and method objects in Smalltalk. This forced me to go deeply into the nature of objects and the fundamental difference between instantiation and generalization. The result 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 could not 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 network* 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 paradigms reported in sections 4 and 5. I believe these paradigms have a value in themselves in addition to being input to the next BabyIDE experiment.

The results of the *fourth experiment* will be decisive. In it, I will 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 and queries. I will need class oriented programming to define the nature of the objects and I will need role model programming to define their interaction. I will also need new debuggers and inspectors to close the loop of figure 22. A great deal of work is needed, and it is probably far in excess of what can be achieved by a single programmer (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 adventurous person who will join me in realizing the BabyIDE vision.

8 REFERENCES

1 James S. Huggin. J. S. First Computer Bug. [web page] http://www.jamesshuggins.com/h/tek1/first_computer_bug.htm

2 Naur, P.; Randell, B. (Ed) Software Engineering. Report on a conference sponsored by the NATO Science Committee. Garmisch, October 1968. Scientific Affairs Division, NATO, Brussels 39, Belgium. p 16.

3 Hoare, C. A. R.: The Emperor's Old Clothes. 1980 Turing Award lecture. *Comm.ACM* vol24-81, 2 (Feb. 1981)

4 Dijkstra, E. D. 1930–2002. Structured programming. E. W. Dijkstra Archive, University of Texas, Document EWD268 [web page] http://www.cs.utexas.edu/users/EWD/transcriptions/EWD02xx/ EWD268.html

5 Dijkstra, E. D. The next fifty years. E. W. Dijkstra Archive, University of Texas, Document EWD1243a. [web page] http://www.cs.utexas.edu/users/EWD/transcriptions/EWD12xx/EWD1243a.html

6 Reenskaug, T.; Wold, P.; Lehne, O.A. *Working with objects. The OOram Software Engineering Method*; Manning; Greenwich, CT, 1996; Out of print. Early version at [web page] http://heim.ifi.uio.no/~trygver/1996/book/WorkingWithObjects.pdf

7 *UML Superstructure Specification, v2.0;* OMG, Needham, MA; 2005. [web page] http://www.omg.org/cgi-bin/doc?formal/2005-07-04

8 Reenskaug, T. *Applications and Technologies for Maritime and Offshore Industries.* In Bubenko, J. A. Jr.; Impagliazzo, J.; Sølvberg, A.; Eds.; *History of Nordic Computing*; ISBN 0-387-24167-1, ISSN 1571-5736 (Print), 1861-2288 (Online); Springer; Boston, MA, 2005; pp 369-390. [Weblink] http://dx.doi.org/10.1007/0-387-24168-X_34

9 Goldberg, A,; Robson, D. *Smalltalk-80. The language and its implementation;* ISBN 0-201-11371-6; Addison-Wesley; Reading, MA, 1983

10 Reenskaug, T. (1979) The original MVC reports. [web page] http://urn.nb.no/URN:NBN:no-14314

11 Birtwistle, G. M.; Dahl, O.; Nygaard, K. *Simula begin.* ISBN 91-44-06211-7; Studentlitteratur; Lund, Sweden, 1973.

12 Reenskaug, T. *Prokon/Plan. A Modelling Tool for Project Planning and Control.* In Gilchrist, B.; Ed.; *Information Processing 77, Proceedings of the IFIP Congress 77*; North-Holland; Amsterdam, Holland, 1977; pp 717-721

13 Wirfs-Brock, R.; McKean, A. *Object Design. Roles, Responsibilities, and Collaborations*. ISBN 0-201-37943-0; Addison-Wesley; Boston, MA, 2003.

14 Wirfs-Brock, R. J.; Johnson, R. E. Surveying Current Research in Object-Oriented Design. *Comm ACM.* **33** 9 (Septeber 1990) 104-124.

15 Reenskaug, T.; et.al. ORASS: seamless support for the creation and maintenance of object-oriented systems. *JOOP*, **5** 6 (October 1992), 27-41.

16 Andersen, E. P. *Conceptual Modeling of Objects. A Role Modeling Approach.* D.Scient thesis, November 1997, University of Oslo. [web page] http://heim.ifi.uio.no/~trygver/1997/EgilAndersen/ConceptualModelingOO.pdf]

17 Gamma et.al. *Design Patterns;* ISBN 0-201-63361-; Addison-Wesley, Reading, MA. 1995.

18 Krasner, G.; Pope, S.T. A Cookbook for Using the Model-View Controller User Interface in Smalltalk-80. *JOOP* **1** 3(Aug./Sept. 1988), 26-49

19 *IFIP-ICC Vocabulary of Information Processing;* North-Holland, Amsterdam, Holland. 1966.

20 Sun Microsystems. *The Java tutorial, What is an object.* [web page] http://java.sun.com/docs/books/tutorial/java/concepts/object.html

21 Fredriksen, K. *UMLexe - UML virtual machine : a framework for model execution.* M.Sc. thesis, University of Oslo, 2005. [web page] http://urn.nb.no/URN:NBN:no-14309

22 Coplien, J. *Multi Paradigm Design for C++*; ISBN: 0-201-82467-1; Addison-Wesley Professional, 1998,

23 Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C., V.: Loingtier, J.; Irwin, J. (1997) *Aspect-Oriented Programming*. In Goos, G.; Hartmanis, J.; van Leeuwen, J. (Eds.) Proc. ECCOP 1977; Springer-Verlag 1997; ISSN 302-9743, DOI 10.1007/BFb0053371, ISBN 3-540-63089-9

24 Hysing, T.; Reenskaug, T. *A System for Computer Plate Preparation*. Numerical Methods Applied to Shipbuilding. A NATO Advanced Study Institute. Oslo-Bergen, 1963.

25 *Squeak*. [web page] http://www.squeak.org/

ACKNOWLEDGEMENTS

The idea and implementation of the Smalltalk stored program object computer is due to Alan Kay, Dan Ingalls, Adele Goldberg and the Learning Research Group at Xerox PARC. UML is the combined result of a great number of people. Taken together, they have unified and documented a large number of powerful concepts for modeling large systems of interacting objects. 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 to Johannes Brodwall for his intelligent support and advice on Java technology. I also thank Øystein Haugen for his thorough commenting of an earlier draft of this chapter.