***Trygve Reenskaug*[1]**

The BabyUML discipline of programming

where A Program = Data + Communication + Algorithms

This article has been published in Software and Systems Modeling.
The original publication is available at
www.springerlink.com The article itself is at http://dx.doi.org/[s10270-006-0005-0-lr].
The manuscript is posted here with permission.

Abstract I want increased confidence in my programs. I want my own and other people's programs to be more readable. I want a new discipline of programming that augments my thought processes. Therefore, I create and explore a new discipline of programming in my BabyUML laboratory. I select, simplify and twist UML and other languages to demonstrate how they help bridge the gap between me as a programmer and the objects running in my computer The focus is on the run time objects; their structure, their interaction, and their individual behaviors.

Keywords. Object-oriented programming – Object oriented methods – Data structures - Object communication - Object algorithms - Latently-typed languages - Stored program object computers

1 I want to be the master of my own software

Let me start by admitting that my programs are not as clear and readable as I would like. They have too many bugs to foster confidence and I feel I am skating on thin ice when I modify and extend them. The ice breaks and I fall through far too often for my liking.

I refuse to accept this state of affairs. Since I cannot easily change my own mental powers, I turn to my tools. Can I improve the concepts and languages I use for thinking about my programs? Can I improve the tools I use to implement them in the first place and modify them in the second?

I believe I know how to program algorithms and declare information models and I have very good tools for performing these tasks. My problems stem from the third aspect of programming; that of specifying communication. *BabyUML* is a new discipline of programming where

a Program = Communication + Data Structure + Algorithm.

BabyUML will allow me to work on a higher architectural level and to create clear and reviewable programs.

I have somewhat whimsically chosen the name *BabyUML*. The English *Baby* was the world's first stored program computer while the target for BabyUML is a virtual, stored program object computer that spans one or more hardware computers. *UML*[2] is a powerful, high level modeling language unifying algorithms, data structures and communication while BabyUML is a programming discipline that builds on selected constructs from UML and other languages.

1.Department of Informatics, University of Oslo, Norway
2.*UML 2.0*. (http://www.omg.org/technology/documents/formal/uml.htm)

1.1 My last comfortable program

The last time my group wrote a program that made me feel comfortable was as long ago as in 1973. The program was memorable because we have never again got a substantial program right the first time.

Our software engineering technology had evolved from its primitive beginnings in 1957 to its culmination in the 1973 program. This program had a database specifying the information model and ensuring referential integrity. It had applications written in FORTRAN with structured design and code. The development process was pure waterfall with peer review of every part. We had plenty of time to keep code and documentation synchronized.

The very satisfactory result was that three out of four subroutines worked correctly at first test while the remaining fourth only had minor bugs. We did not find any errors during system integration and subsequent usage. Indeed, we would have been very surprised if we had found any errors in the program's life time.

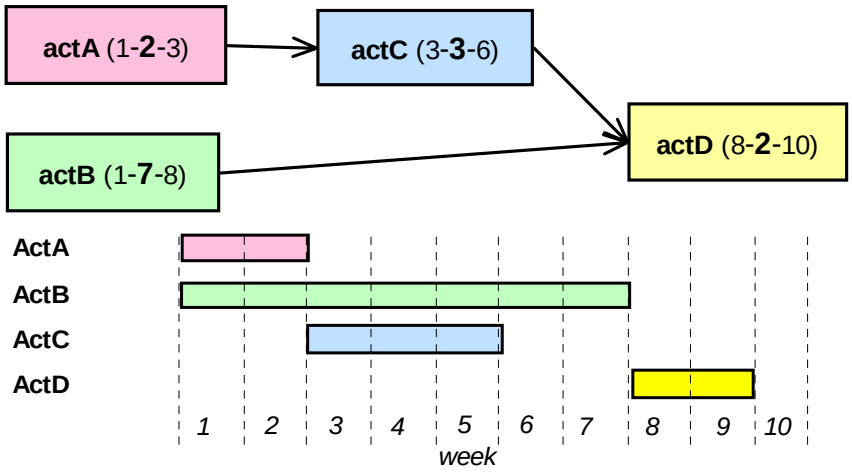
1.2 A New world with new challenges

The world around us changed dramatically and the success was never repeated. We never again had the time and money needed to maintain updated documentation. We never again experienced stable specifications giving us time to contemplate on a clean and simple solution. Systems were never finished; continuous system evolution was needed to match ever changing requirements and insights. The closed systems of the past were replaced by open systems that were influenced by the systems around them, and, in their turn, influenced them. Users were becoming more computer literate and wanted clear mental models of their computer based tools. Distributed systems were needed to give different user communities ownership of their information.

1.3 An example

I'll illustrate my problem and describe my solution with a simplified project planning example. A project is split into *activities* where each activity represents a certain piece of work. An activity is characterized by its *name*, its *duration*, its *predecessor* activities that need to be finished before this activity can start, and its *successor* activities that can start when this activity is finished. The example activity network and the result of frontloading is shown in Fig. 1 where each activity is described by *name (early start - duration - early finish)*.

Fig. 1: A simple activity network.



1.4 The problem

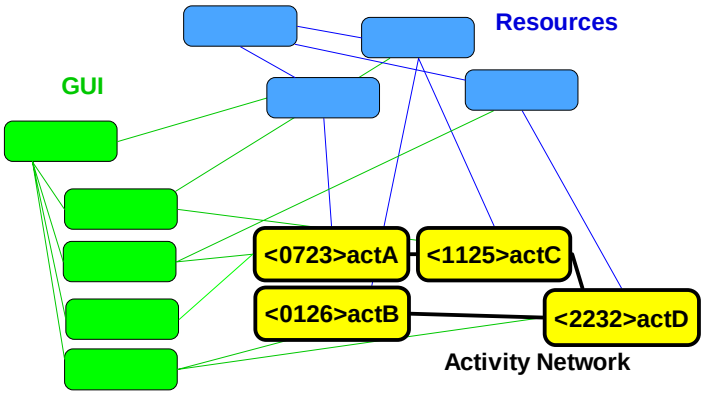
1973 marked our transition into a new and more complex world where *communication* added a new dimension to our challenges. The answer was *object orientation*; a technology that was created by Nygaard and Dahl that helped me master complexity. It enabled me to tackle more complex problems and my solutions were markedly more flexible. But object technology has proven to have its limitations. After more than 10 years of development, I found that our group was working on a program of more than a quarter million lines of Smalltalk code. Our group was very stable. Most of us had a good understanding of the system. There were few bugs, and we extended the program with great confidence. But there was a downside. Newcomers to our group found it very hard to understand its elaborate architecture and numerous conventions. And today, I find the program code almost unreadable since I have forgotten most of its internal features.

There is clearly a limit to the complexity I can master with my current OO technology. I find that as the size of the system grows, it gets more and more difficult to see the whole in the mass of detailed code. I claim that I need higher level programming constructs to master the increased complexity of my systems.

Figure 2 illustrates some application objects for project planning and control with activity networks. The activity objects are shown bottom right with heavy outlines. The idea is that planning is realized by negotiation; internally between the activity objects themselves and externally between activity objects and their required resources (manpower, machinery, etc.) The technicalities of user interface are separated from the domain objects; the GUI objects are shown to the left in the figure.

This application is typical of the style I have been using for years. It tends to give medium sized objects, distributed logic and distributed control leading to a large number of crisscrossing links. An activity uses a certain resource; let the activity object negotiate directly with the resource object to establish a mutually acceptable schedule. A figure on the computer screen represents a certain activity; let the figure object interrogate the activity object to determine how it is to be presented, and let the activity object warn the figure object of significant changes. The result is an unmanageable bowl of spaghetti.

Fig. 2: A typical application.



Every object is an instance of some class written in a language such as Simula, Java, or Smalltalk. The structure and domain logic is distributed into the methods of the classes with their superclasses, effectively fragmenting the bowl of spaghetti into a dish of noodles.

> *My programming language and my programming style tend to make large systems look like bowls of spaghetti. Extreme fragmentation tends to chop each strand into noodles. The system as a whole is nowhere to be seen.*

The most interesting part of a system is the communication that takes place in the space between the objects. This is where my current style and language fail miserably. I need to replace the dish of noodles with a clear and manageable declaration of structure. I need to pull essential information out of the noodles to define object interaction in explicit and readable code. I simply need to separate the code that defines the whole from the code that defines the details.

The increasing popularity of object modelling languages such as UML, MOF, and MDA suggests that my needs may be shared by others. Alan Kay once said that *an operating system is what the language designers omitted to include in their language*. We could paraphrase this and say that *a modeling language is what the language designers omitted to include in the programming languages*.

1.5 BabyUML: A Glimpse of a solution

The aim of the BabyUML project is first to identify the core concepts I need to regain control of my systems. Second, I want to create a run time environment supporting these concepts. Third, I want to define the languages and interactive development environments that lets me experiment with the new concepts and facilities. My vision is that BabyUML will let me write significant programs that make me feel as comfortable as I felt with the memorable 1973 program.

Baby UML introduces four fundamental extensions to the object model as supported by the current programming languages:

- *A BabyUML Component* disentangles the spaghetti by encapsulating a number of related objects. It provides hierarchical partitioning of the object space, furnishing important leverage to a strategy of *divide and conquer.* A BabyUML component looks like a single object when seen from the outside and is characterized by its *provided and required interfaces*. Inside, the component contains a system of interacting *Member Objects* that collectively implement the provided interfaces and use the required ones.
- *Data.* A component Member Object Declaration makes essential structure information explicit. I code a *conceptual schema* for the Member Objects and their associations, naming essential elements and guaranteeing referential integrity.
- *Communication.* I specify component Member Object Interaction by extracting essential information from the noodles, thus making the overall interaction explicit. Each operation declared in the provided interface is realized by collaborating objects where each Member Object plays a specific *role* in the interaction.
- *Algorithm.* I code the detailed behavior of the Member Objects in the methods of the appropriate implementation classes.

I describe each of these extensions below and end with describing the BabyUML laboratory.

The laboratory is *a Stored Program Object Computer*. Binary and source code program elements such as classes and methods exist at run time as regular objects. The same applies to compilers, inspectors and debuggers so that programs are largely self-documenting and can be modified dynamically.

2 The BabyUML component

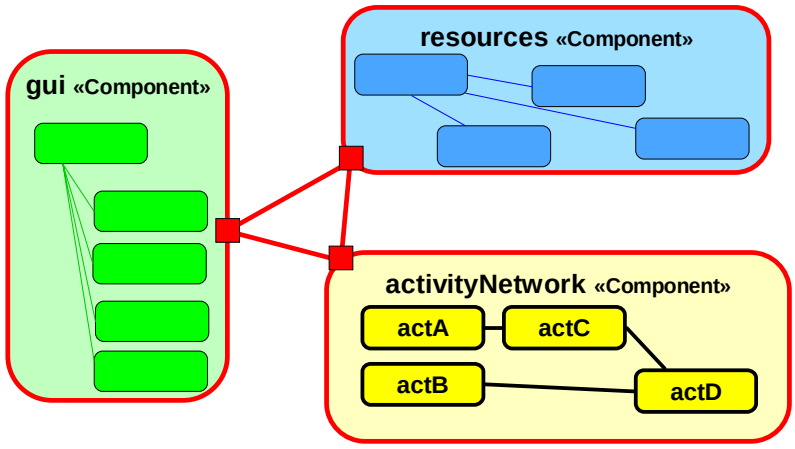
The first step is to bring discipline to the universe of interlinked objects. My current Smalltalk system contains some 300,000 objects; I need some kind of modularization in order to divide and conquer.

UML 2 defines the Component metaclass as a kind of Class. It specifies a model element that owns and encapsulates a number of *member elements* and is characterized by its *required* and *provided interfaces*:

A babyUMLComponent is an object that encapsulates other objects. The babyUMLComponent looks like a regular object seen from its environment; objects and components can be used interchangeably. Inside, we find a bounded structure of interacting *Member Objects*.

Figure 3 illustrates how the spaghetti of Fig. 2 is replaced by a simple structure of three interacting components. The babyUMLComponent notion is recursive; I can organize several hundred thousand objects in a component structure so that I can deal with a manageable number at each level.

Fig. 3: A Component is an object that encapsulates other objects



2.1 Clear advantages

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

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

2.2 No leverage without rigidity

A rubber crowbar can do almost anything apart from providing leverage. *There is no leverage without rigidity.* The challenge is to provide leverage where it does most good and keep rigidity where it does not hurt. I have chosen to limit my freedom as a programmer by insisting on a clean component structure. I gain readability and reliability. I loose the fun writing smart and obscure code. I may also loose some performance. (Anybody remember the time when experts declared relational databases to be too inefficient for practical use?)

The big question is if I can live with this very strong grouping discipline. New systems can be conceived in a component architecture from the outset. It may not be easy to regroup my current objects into components because they do not exhibit a clear and bounded grouping. But there should be no problems with legacy systems because they can be encapsulated within BabyUML components and look like ordinary objects in their environment.

In the early stages, I am likely to protect my old way of thinking by inventing reasons why components is a bad idea because I miss the freedom to send any message anywhere at any time. I expect it will take some time and effort to redirect my brain to think in terms of disciplined components rather than the distributed debating societies I am used to. But I am confident that I will grow to like components and wonder how I could ever do without them. I will become better at mastering complexity and I will avoid the

bugs where a change in one corner of a system causes a failure in the opposite corner. A similar development has happened to me before; I no longer miss the once cherished goto-statement

I may perceive reduced performance when I force messages to pass through the component interfaces. But on the other hand, the clear system architecture may reveal bottlenecks and opportunities for optimization. Further, a clean and precisely described structure should facilitate automatic optimization technologies.

It is commonly believed that small, highly distributed objects simplify system evolution because changes will be localized to single classes. But this is not always true. There will always be changes that cross the class boundaries. A clean and explicit structure is the best basis for handling changes.

3 Data: Member Object declaration

There can be many kinds of BabyUML components and the different kinds can have different internal structures and be programmed in different languages. Disparate components can be mixed within the same system since their differences will be hidden within their boundaries. I will here describe the currently most powerful and complex kind, the *Declarative Component*. I here gain leverage by imposing a strict discipline on the Member Objects, the links between them and their interaction.

A snippet of class definition code is shown in Fig. 4. (Here coded in Squeak[1]). It defines the Activity1 class with five instance variables and three methods. Let me pretend that I see this code for the first time and try to disentangle it.

Fig. 4: A very simple class.

```
Object subclass: #Activity1
    instanceVariableNames:
        ' name duration preds succs earlyStart '
------------
Activity1>>initialize
    name := 'Unnamed activity';
    preds := Set new.
    succs := Set new.
------------
Activity1>>addSuccessor: anActivity
    succs add: anActivity.
------------
Activity1>>addPredecessor: anActivity
    preds add: anActivity.
```

The initialize method tells me that preds and succs are Sets. But how are the objects structured? Knowing that this program is supposed to simulate an activity planning network, I quickly jump to the conclusion that the activities are linked with binary relationships so that all the succs of an activity object has this object as one of its preds.

I suspect that there is something fishy somewhere and want to review the code in detail. How do I know that preds and succs always come in pairs? How do I know what happens if I change the name of an activity? Which objects send the structuring messages, when and why? Are there any other methods that modify succs and preds? I have to read many pages of code to find the answer. If I still remember the question.

It is, of course, easy to see how to make this code safe. Let addSuccessor: and addPredecessor: be the only methods that change succs and preds and let them ensure the binary relationship. Easy, but I didn't write the code. I find 15 subclasses of Activity1. Some of them actually provide very cool features with very sophisticated ways of building the activity network. Some of them write directly to the succs and preds variables ...

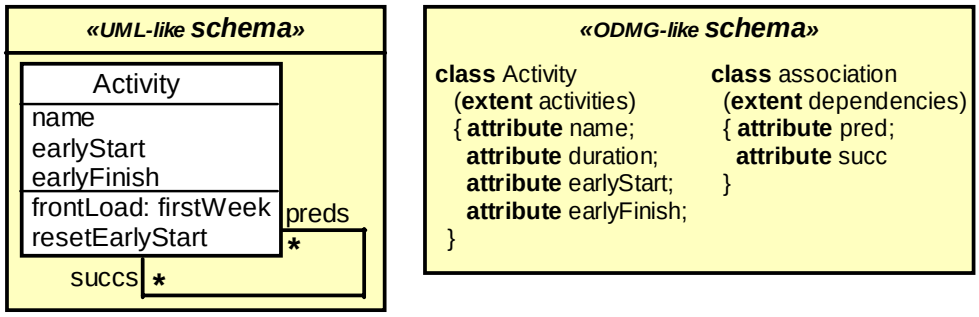
---

1. *Squeak*. http://www.squeak.org/

3.1 Declaring the Member Object structure

It would have helped if the class diagram of Fig. 5 had been attached to the code and if I could trust that all programmers had conformed to it. But it could be out of date, or some hidden detail in the code could unintentionally violate it.

Fig. 5: One simple schema, two languages. .



In the Declarative Component, the schema *is* the code. It is shown in two alternative languages in Fig. 5. One language is a derivative of the UML class diagram, the other is based on ODMG[1]. I have not declared types in either code; this reflects that the BabyUML typing system is still an open issue. The ODMG version names the extent; here it gives a name to the set of all instances that exist in a given component. A UML alternative is to use the UML Object Constraint Language to find the set of instances.

I delegate the creation and maintenance of the interlinked Member Objects to a new "micro database" object; I name it babyBase. The schema is always up to date since it *is* the code. Clients and/or subclasses can no longer do cool and fancy things in novel and innovative ways. They will have to explicitly modify and query the structure by messages to the babyBase object. The code in Fig. 6 illustrates the difference. To the left, traditional code where preds is an instance variable that a reader has to check carefully . To the right, the meaning is explicit and there can be no hidden surprises.

Fig. 6: Snippet of code using the Member Object declaration of Fig. 5.

"Traditional OO programming"

```
frontload: firstWeek
    earlyStart := firstWeek.
    preds do:
        [:predecessor |
        earlyStart
            := earlyStart max: predecessor earlyFinish]
```

"With declared Member Objects."

```
frontload: firstWeek
    earlyStart := firstWeek.
    (babyBase predecessorsOf: self) do:
        [:predecessor |
        earlyStart
            := earlyStart max: predecessor earlyFinish]
```

Whenever I try to read other people's programs, I struggle to find the answer to three critical questions: *What are the objects? How are they interlinked? How to they collaborate?* Extending the code with a declaration of the Member Object data structure gives me an initial answer. But I will have to extend the code with a specification of the system dynamics to get a complete answer. This is the theme of the next section.

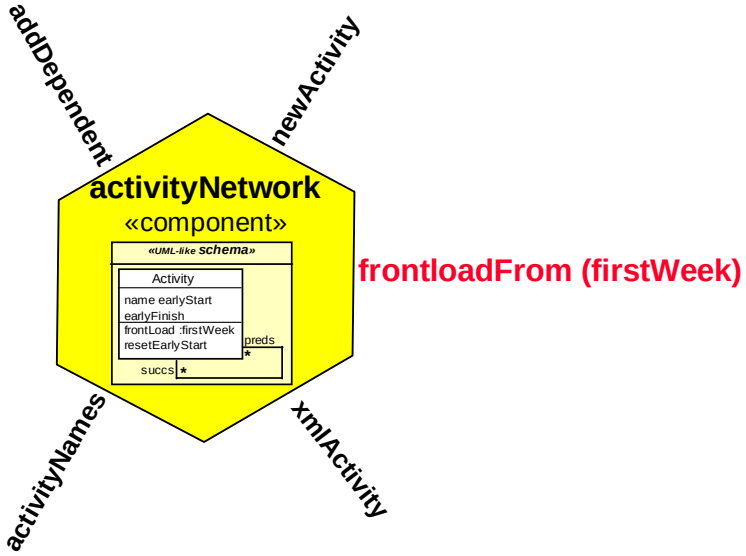
4 Communication: specification of component behavior

I am now ready to realize the component's provided operations. The idea is to work with the component in different *projections* as illustrated in Fig. 7. Each projection contains the code that specifies how the component realizes one of the provided operations.

A basic operation on an activity network is *frontloading* that computes earlyStart for each activity when the start of the project is known. (Other important network operations are ignored here. Examples are *backloading* that computes the *lateFinish* time for each activity when the finish time for the project is known, and operations for the handling

of overlapping activities.). I have chosen this operation as my example and use it to illustrate high level coding of object systems.

Fig. 7: One component projection for each provided operation.



Different kinds of components can use different languages for coding how the Member Objects work together to realize the component's provided operations. I will here illustrate how I code a DeclarativeComponent using a mixture of graphical and textual languages:

1. I code the interaction as a modified UML sequence diagram.
2. I define a collaboration where the roles are bound to the timelines of the sequence diagram.
3. I query the component's conceptual schema to find the objects currently playing the different roles.
4. I finally code the object's methods in the preferred third generation programming language.

4.1 Member object interaction code

I see the component as an orchestra. The Member Objects are the performers. A *maestro* directs the performance; controlling when the members shall execute their pieces.

Fig. 8: Frontloading sequence diagram.

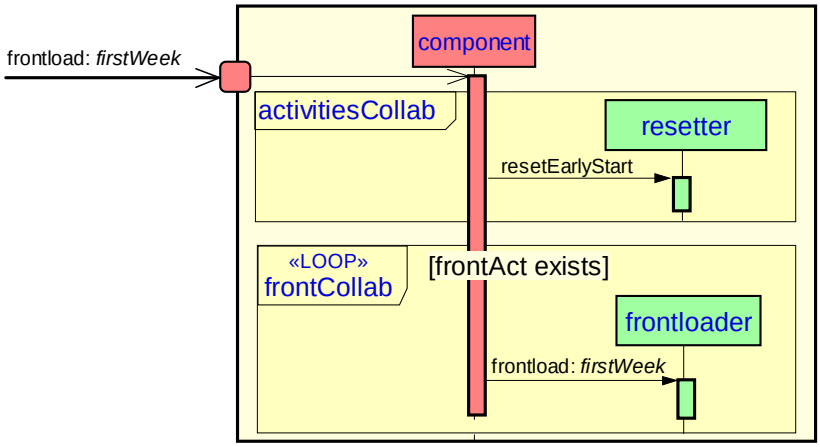


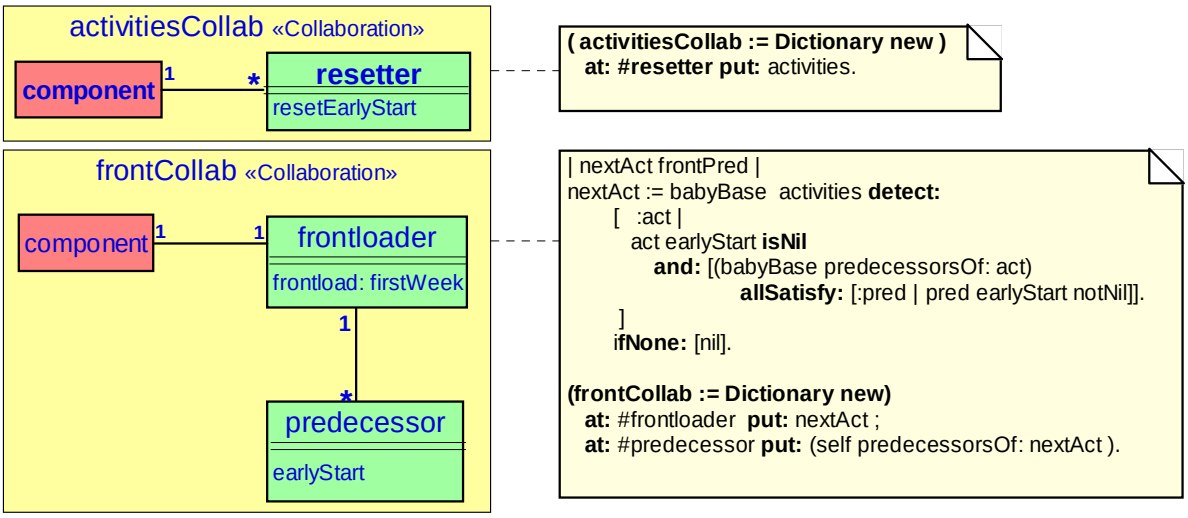
Figure 8 shows the code for my component's implementation of the frontLoad: operation. The code looks like a *UML Sequence Diagram*, but I have adapted its semantics to serve my purposes. BabyUML interprets the diagram as a sequence of two fragments:

- In the activitiesCollab fragment, the maestro sends the message resetEarlyStart to all objects playing the resetter role.
- In the frontCollab fragment, the maestro sends the frontload: firstWeek message to an object that plays the frontloader role. This fragment is a loop that is executed repeatedly as long as any object is playing the frontloader role.

4.2 Binding timelines to collaboration roles

Each fragment in Fig. 8 is executed in the context of the corresponding collaboration shown in Fig. 9i and each timeline is performed by an object playing the corresponding role. There are two collaborations named activitiesCollab and frontCollab in this example.

Fig. 9: Frontloading collaborations.



(i) Collaborations *(ii) Queries*

4.3 Binding collaboration roles to Member Objects

An object can play many different roles and a role can be played by many different objects. This means that a class can implement the features needed to play a set of roles and another class can implement the features needed for another set; some roles being the same and some different.

The key issue is to bind roles to objects. To me, the breakthrough came when I realized that the roles of a collaboration can be found by queries on the conceptual schema. Queries are expressed in a query language such as the Object Query Language of ODMG or the UML Object Constraint Language. The queries can alternatively be expressed in the concrete implementation language as shown in Fig. 9ii.

In the top query, I create the activitiesCollab context where I bind the role name resetter to the set of all Activity instances in the current component. In the bottom query, I create the frontCollab context where the frontloader role is bound to an unloaded activity where all the predecessors are loaded, and the predecessor role is bound to all the predecessors of the current frontloader object.

The BabyUML component architecture discussed here corresponds closely to the three schema architecture used in the database community[4]:

- The *conceptual schema* (fig. 5 on page 7) defines the component universe of discourse with all its legal members and associations.
- The *external schemas* (Fig. 9i) define the collaborations, i.e., the roles played by the objects that realize the component's provided operations. Their external mappings bind them to the conceptual schema by suitable queries. (Fig. 9ii).

- The *internal schema* describes how the conceptual schema is actually implemented in the component, i.e., the Member Object implementation classes.

4.4 Algorithms: class-centered programming of the details

Figure 9 shows that the Member Objects playing the various roles need to understand the following messages: resetEarlyStart, earlyFinish, and frontload: firstWeek. I now get to the details and code the corresponding methods. There is still plenty of scope for differentiation; different classes implementing the roles can have different ways for handling the messages. The conceptual Activity objects can be components or simple instances of classes or a mixture of both. But it will all be done within strict limits; the Member Objects must conform to the schema and the Member Object interaction will be as specified in the collaborations and interactions.

The methods are coded in the context of the corresponding collaboration so the Member Objects are accessed with their role names. The methods are shown in Fig. 10.

Fig. 10: Implementing the details.

```
Object subclass: #Activity2
    instanceVariableNames:
        ' name earlyStart duration '
------------
Activity2>>resetEarlyStart
    earlyStart := nil.
```

```
------------
Activity2>>frontload: firstWeek
        earlyStart := firstWeek.
        (self context at: #predecessor) do:
            [:pred |
            earlyStart := earlyStart max: pred earlyFinish ].
```

Notice that context is now the dynamic dictionary binding names to query results as illustrated in Fig. 9ii. (It turns out that it actually lives on the stack).

5 The BabyUML laboratory

The research method of the BabyUML project is essentially experimental. The *BabyUML laboratory* lets me try out different architectures and paradigms in order to better support my thought processes and improve the self-documenting properties of my programs. The primary focus is on the objects; their structure, their interaction and their features. The laboratory is based on Squeak, a dialect of Smalltalk[3]. Smalltalk is ideal for the purpose with its pure object based execution model, its transparency and flexibility.

Squeak was chosen because its friendly licence promotes sharing the BabyUML programs and systems. Other reasons are that there is a very active and open community around Squeak, and that there are a number of innovative ideas available as packages that I can file in and modify as parts of BabyUML.

5.1 BabyUML: A stored program object computer

Squeak is a virtual, stored program, object computer as illustrated in Fig. 11. *Object*, because its smallest unit of data is the object. All data are represented as objects; even booleans, numbers and characters. *Virtual*, because it is realized in software by the Squeak Virtual Machine (VM) that manages the objects and executes the programs. *Stored program* because classes, methods and even the execution stack frames are seen and manipulated as regular objects. A program can operate upon any object in the VM, including the program itself. This means that compilers, inspectors and debuggers likewise are regular objects at run time.

Fig. 11: The BabyUML laboratory is embedded in the Smalltalk object space

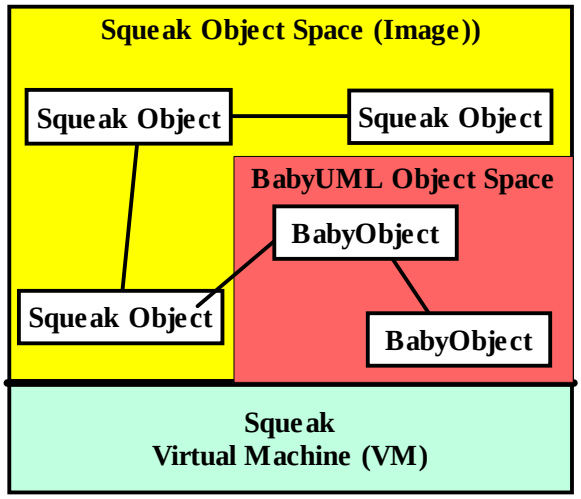


Figure 11 illustrates the BabyUML implementation. The Baby objects have their own classes, metaclasses, and metametaclasses. I can and do build my own versions of these objects to experiment with different ideas. These versions exist and operate together with the originals and can interoperate with them because they conform to the simple VM conventions.

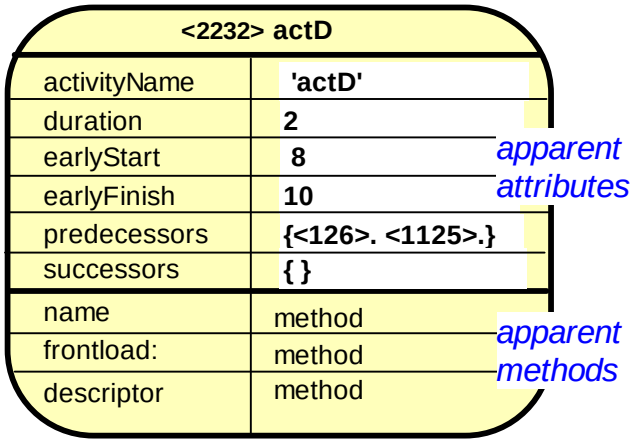
5.2 The BabyUML Object Notation

An object is an entity that encapsulates state and behavior with a unique identity and that is accessed through its message interface. BabyUML uses three different notations for objects: the *encapsulated object*, the *conceptual object*, and the *implemented object*.

The *encapsulated object notation* is illustrated in Fig. 2 in section 1.4. It shows objects as rounded rectangles, possibly with their identities and names. (I use the rounded rectangle to distinguish it from the *UML classifier* which is a very different concept).

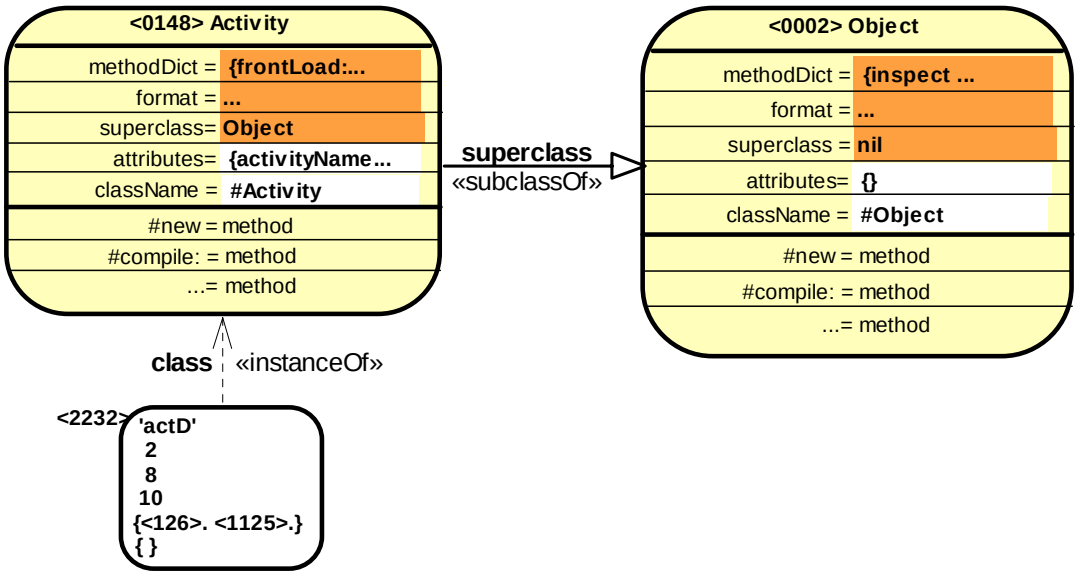
The *conceptual object notation* is illustrated in Fig. 12. It shows the object features without being cluttered with implementation details such as classes and superclasses.

Fig. 12: The conceptual object notation.



Every object is an instance of a class, and that class is the subclass of some other class. The *implementation notation* shows the instance and its class objects as illustrated in Fig. 13.

Fig. 13: The implementation of an object.



The objects are as follows:

- The instance itself has objectID=<2232> and stores the values 'actD', 2, 8, 10, {<126>. <1125>.}.
- Every object is an instance of a class and has a link to it. <2232> is an instance of class Activity, stored 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 <2232> attributes are the concatenation of the attributes attributes of the class and all its superclasses.
- The <2232> methods are the union of the methods defined in the methodDict attribute of the class and its superclasses.
- Note that the frontload: - method is stored in the class <0148>Activity. The inspect-method is stored in the class <0002>Object. Both of them are visible to the collaborators of the <2232> object as bona fide operations on that object. *The actual factoring of methods along the superclass chain has no semantic significance and is in the nature of a comment.*

Also note that the class objects have their own attributes and methods. In this particular implementation, the class objects respond to the new and compile: messages. Different classes may, therefore, have different factory methods and compile their methods from code expressed in different languages.

Even a class object is an instance of some class, called its *metaclass*. It is a rich source of confusion that the implementation of the <2232>actD object is defined in the class objects shown in Fig. 13, while the features of the <0148>Activity class object is stored in *its* metaclass with superclasses. Hence the value of the conceptual view that shows the semantics of the object and hides the complexity of the implementation.

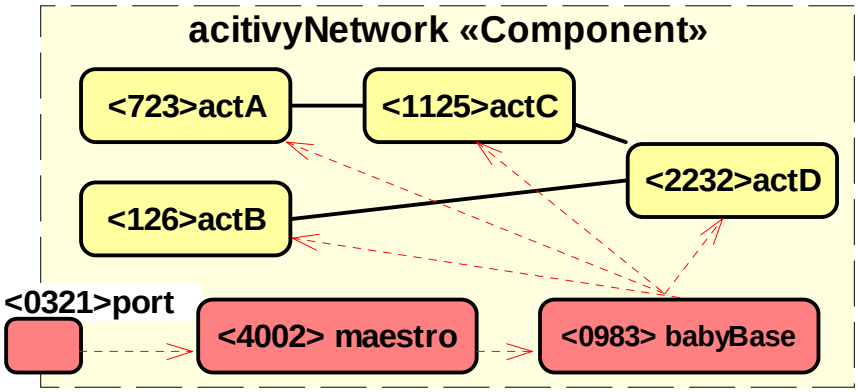
6 Further work

In section 2, I discussed the BabyUML component as an object that encapsulates a cluster of Member Objects. I will now show that it also encapsulates its own management objects.

Figure 14 shows the objects in my first experimental implementation of the ActivityNetwork example component. The Member Objects represent the activities. In addition, there are three special objects that

relate to the component as a whole. The babyBase object manages the Member Objects, the maestro object orchestrates Member Object interaction, and the port is the faēade object seen in the environment.

Fig. 14: The objects of the activityNetwork component.



The objects of Fig. 14 are instances of a number of different classes; in this experiment they happen to be DeclarativeActivity, DeclarativePort, DeclarativeMaestro, and DeclarativeBase . Instantiation and linking is done in an external test program.

A striking feature of this first example is that there is no run time object representing the component as a whole. Likewise, there is no class that acts as a factory object for new, empty network components. Inspectors and debuggers show the bare objects and stack frames; the component is a feature of the documentation only.

My next experiment will be to define a component class that can act as a component factory for the coordinated instantiation of port, maestro, and database objects. The instances of the component class itself will be responsible for the high level handling of components in inspectors and debuggers. These objects could also be responsible for enforcing privacy and security.

The features of an object are defined by its *class*. The features of a class object are defined by *its* class; the *metaclass*. Languages such as Java have a single metaclass; all classes have the same features. In regular Smalltalk, classes and metaclasses come in pairs; the metaclass being the logical place to store static variables and methods. UML 2.0 defines two different metaclasses: The *Class* and the *Component*.

One of the early BabyUML experiments was to augment the Smalltalk class architecture with a new set of metaclasses that were instances of a common metametaclass. This proved that the Smalltalk metaclass architecture is a feature of the class library; other architectures can coexist with the regular one. Different metaclasses can support programming in different languages. BabyUML can, therefore, support a multi-paradigm approach[2] combining algorithms, communication and data structures with different languages for these paradigms.

A by-product of my next experiment will be to explore whether the component class should be a regular class or if it will be more convenient to make it an instance of a new component metaclass.

Once the basic architecture has been selected, the class hierarchy will be refactored to create one or more metamodels in the sense this term is used in UML.

7 Conclusion

I started programming in 1957. The language was the computer's binary instruction set; the only organizing principle was the subroutine. I soon learned that this algorithmic approach had to be subordinated an information model. By 1961, we had a central database surrounded by a number of applications[5].

By the early seventies, it was evident that the databases had to be subordinated a distributed component-based architecture[6]. Object orientation and the Simula language had been invented by Nygaard and Dahl at the Norwegian Computing Center, just across the yard from my office. It seemed reasonable to base a project for the distributed planning and control on object orientation.

An attempt to use Simula failed for two reasons. First, because I didn't find a way to make Simula objects persistent. Second, the Simula typing system broke the object encapsulation property because I had to declare every variable with the implementation (class) of the referenced objects. For a variety of reasons, the planning project died before I got a working prototype on the air. The ideas survived, however. My first, primitive interaction diagram was published at an IFIP conference in 1977[7]. The Model-View-Controller pattern was a result of thinking in terms of objects rather than classes.[1]

The concept of *role modeling*[8] was a major step forward, focusing on the roles objects play in an interaction independently of their implementation classes. Our UML submission to the OMG was based on role modeling and parts of it found its way into the standard under the name of *collaborations*.

Fig. 15: Main Concepts.

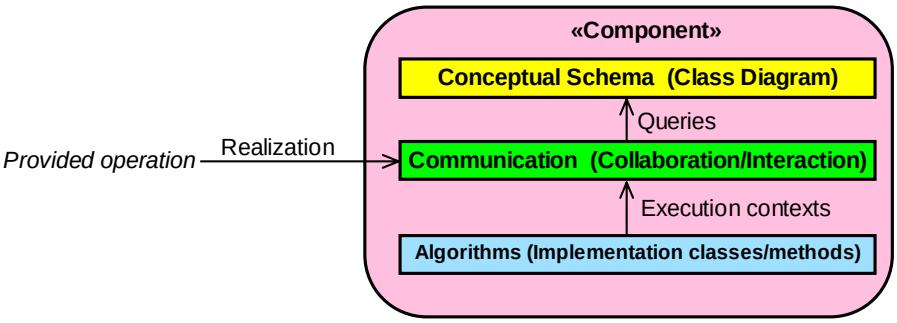


Figure 15 illustrates how the new declarative component encapsulates the declaration of data structures with the specification of communication processes and the definition of algorithms. This might look like an obfuscator at a first glance; but in reality it is a clarifier. In my traditional programming style, the whole is hidden in the details. Here, the whole is extracted and made explicit. This will simplify my thought processes and facilitate clean and readable programs that can be mastered by me, future users, and future maintainers.

My initial and fragmentary experiments look promising; but considerable work will be needed before I can regain the control I have been missing for the past 30 years.

1.Reenskaug: *Thing - Model - View - Editor, an Example from a planningsystem.* Xerox PARC note, May 1979 http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf

References.

[1]	Cattell, Barry: The Object Data Standard: ODMG 3.0. Academic Press, London, 2000. ISBN 1-55860-647-4 (http://www.odmg.org/)
[2]	Coplien, James: *Multi Paradigm Design for C++*, Addison-Wesley Professional, 1998, ISBN: 0-201-82467-1
[3]	Goldberg, Robson: *Smalltalk-80, the language and its implementation.* ("The Blue Book"). Addison-Wesley, Reading 1983. ISBN0-201-11371-6
[4]	Hay, David: *What Exactly IS a Data Model?* DM Review Magazine, February 2003
[5]	Hysing, Reenskaug: *A System for Computer Plate Preparation*. Numerical Methods Applied to Shipbuilding. A NATO Advanced Study Institute. Oslo-Bergen, 1963.
[6]	Reenskaug: *Administrative Control in the Shipyard.* ICCAS conference, Tokyo, 1973. (http://heim.ifi.uio.no/~trygver/1973/iccas/1973-08-ICCAS.pdf)
[7]	Reenskaug: *Prokon/Plan. A Modelling Tool for Project Planning and Control.* IFIP Congress, Toronto, Canada, 1977. http://heim.ifi.uio.no/~trygver/1977/Prokon/IFIP-Prokon.pdf
[8]	Reenskaug et.al.: *Working with objects. The OOram Software Engineering Method.* Prentice-Hall 1996. Early version scanned at (http://heim.ifi.uio.no/~trygver/1996/book/WorkingWithObjects.pdf)

Trygve Reenskaug is professor emeritus of informatics at the University of Oslo. He has 40 years experience in software engineering research and the development of industrial strength software products. He has extensive teaching and speaking experience including keynotes, talks and tutorials. His firsts include the Autokon system for computer aided design of ships with end user programming language, structured programming, and a data base oriented architecture from 1960; object oriented applications and role (collaboration) modeling from 1973; Model-View-Controller, the world's first reusable object oriented framework, from 1979; OOram role modeling method and tool from 1983. Trygve was a member of the UML Core Team and was a contributor to UML 1.4. The goal of his current research is to create a new, high level discipline of programming that lets us reclaim the mastery of software.