Towards a New Discipline of Programming
with the BabyUML Language Laboratory

Trygve Reenskaug, Department of informatics, University of Oslo, Norway

VERY EARLY DRAFT 2005-06-01

Abstract

1973 marked the end of an era and the beginning of a new in my career as a computer programmer. The first era culminated with me writing the last program of a series that gave me a comfortable feeling. The new era came about because we moved from stable, closed systems to continuously changing, open systems. New problems, new challenges. I have still not mastered them; I still cannot build a significant program that is “so simple that there are obviously no deficiencies”.

There is a widespread belief that current software complexity is an unavoidable consequence of the complexity of the requirements and that we have to tolerate deficient software. I claim that this is merely a myth, and that our problems can be solved with adequate programming concepts, languages and tools that leverage the communication aspects of our systems.

It is hard to do sums with roman numerals. It is equally hard to build communicating systems with current tools. BabyUML is a laboratory for experimenting with new programming disciplines and languages. The focus is on communicating objects rather than static classes. The laboratory is used to explore the specification of hierarchical structures of collaborating components and to explore how to make the component essence explicit and visible, hiding the details within the component members. The goal is to disprove the complexity myth and to help me feel comfortable with my future programs.

i: Abstract :

Current information systems have very broad requirements including the need for openness and distribution coupled with privacy and security. Continuous system evolution is needed to adapt them to a changing environment. The BabyUML laboratory is an environment for experimenting with new ways of building comprehensible and secure systems with transparent and readable code. BabyUML is a *stored program object computer*. It is implemented in Squeak, a dialect of Smalltalk. All program elements including classes, metaclasses and compilers exist as tangible objects, merging program build time into run time. The Smalltalk insistence on pure object orientation invites us to change our focus from the classes to the objects. We supersede the Smalltalk notion of programs with new high-level programming constructs gleaned from UML and transpose them to fit in the object space. The result is a set of high-level constructs that will help create clear solutions to complex problems.

BabyUML may be seen as a contribution to a possible UML 3.0; a unified, high level, and extensible family of programming languages for distributed, stored program object computers.

In the tutorial, we will discuss the fundamental concepts of BabyUML and illustrate them with the implementation of a concrete example.

1973 marked the end of an era and the beginning of a new in my career as a computer programmer. The first era culminated with me writing the last program of a series that gave me a comfortable feeling. The new era came about because we moved from stable, closed systems to continuously changing, open systems. New problems, new challenges. I have still not mastered them; I still cannot build a significant program that is “so simple that there are obviously no deficiencies”. 1

There is a widespread belief that current software complexity is an unavoidable consequence of the complexity of the requirements and that we have to tolerate deficient software. I claim that this is merely a myth, and that our problems can be solved with adequate programming concepts, languages and tools that leverage the communication aspects of our systems. 1

It is hard to do sums with roman numerals. It is equally hard to build communicating systems with current tools. BabyUML is a laboratory for experimenting with new programming disciplines and languages. The focus is on communicating objects rather than static classes. The laboratory is used to explore the specification of hierarchical structures of collaborating components and to explore how to make the component essence explicit and visible, hiding the details within the component members. The goal is to disprove the complexity myth and to help me feel comfortable with my future programs. 1

i: Abstract : 1

Part A: Introduction 5

A.1: Preface 5
i: The New Nature of Programming 5
ii: The Need for a New Generation 6
iii: Acknowledgements 6
iv: Outline 7
A.2: Three Generations of System Architectures 8
i: CPU-Centered Architecture 8
ii: Storage-Centered Architecture 8
iii: Communication-Centered Architectures 9
A.3: Introduction to BabyUML 10
i: The BabyUML Stored Program Object Computer 10
ii: The BabyUML Laboratory is Embedded in Squeak 11
iii: BabyUML Layered Architecture 12

Part B: Example Objects 14

B.1: The Simple Object 14
i: The Encapsulated Projection 14
ii: The Conceptual Projection 15
B.2: The Simple Component 17
i: A Component Encapsulates Objects 17
ii: Encapsulated Projection of a Simple Component 18
iii: A Conceptual Projection of the Example Component 18
B.3: The Mastered Component 19
B.4: The Declarative Component 22
i: Storage-Centered: A Micro “Database” Holding the Members of the Component 23
ii: Communication-Centered: Maestro-controlled interaction 24
iii: Mapping roles to members 25
iv: Member interaction 26
v: CPU-Centered programming of the details 26

Part C: The Classes 27

C.1: The SimpleClass 27

i: The black box Object Descriptor 27
ii: The object implementation 27
iii: Browsing the Object Descriptor 28
C.2: Component Merge 30
i: The Object Descriptor described as Merged Components 30
ii: The sampleNet Specified as Merged Components 31
C.3: The Simple Component 33
i: The Simple Component Descriptor 33
ii: Implementation of the network component class 33
iii: Excerpts of the sampleNet code 34
Part D: The Metaclasses 35
D.1: The MetaSimpleclass 35
i: The implementation of the Activity class object 35
ii: The instantiation and inheritance relationships 35
D.2: The MetaSimpleComponent 37
D.3: The MetaMasteredComponent 37
D.4: The MetaDeclarativeComponent 37
Part E: Windup 38
E.1: Conclusion 38
E.2: Further work 39
References 40
Appendix 1: The Declarative Component implementation 42
i: activityNamed: aString duration: anInteger 43
ii: addDependentFrom: actName1 to: actName2 44
iii: frontload: firstWeek 44
iv: activityNames 45
v: activityDescriptorFor: actName 45
Appendix 2: Component Merge 46

Part A: Introduction

A.1: Preface

Nusse was the first Norwegian modern computer, it started its operation in 1953. It was a binary computer; all programs and data had a uniformly binary representation. Its memory sported a grand total of 16,384 bits organized in 512 words of 32 bits each. It was a stored program computer; a program could operate on any word in memory, including the program itself.

I joined the Nusse team in 1957. I learnt to program sitting in front of the CPU console watching the registers, changing their bits one by one and stepping through the program instructions. I was in intimate contact with the CPU and had to bridge the gap to the problem domain in my head. There were no aids to program structuring. Program and data words were more or less haphazardly sprawled all over memory. I soon learned that 512 words was more than my brain could master. A typical symptom was that an innocuous change in one corner of a program caused a catastrophic failure in the opposite corner.

Our software engineering savvy improved over the years and culminated in 1973 . This was the last time I experienced a program that made me feel comfortable. The problem solved was nontrivial. We worked in a waterfall fashion on its specification, design and database schema until everything was clear, simple and well understood. We then coded one subroutine at the time, using peer review to ensure its correctness before we tested it. The unit tests revealed one error per four hundred lines of code, system tests and subsequent use revealed none. The completion of this 1973 program gave us great satisfaction and we would have been very surprised if it hadn't fulfilled our expectations throughout its lifetime.

i: The New Nature of Programming

The 1973 program marked the end of an era because the nature of our work with computers underwent a fundamental change at that time:

- *Evolution*. The static user requirements of the sixties were replaced by tight loops where new programs caused new user work procedures that in its turn lead to new requirements. It was no longer practicable to maintain specification and design documents synchronized with the rapidly evolving programs.
- *Open systems*. The systems of the sixties were closed, monolithic, database centered systems. We now saw open systems of distributed components with no clear boundaries between them. In the sixties, an engineer who wanted to retrieve some data from the design system and feed them into the materials management system needed two computer terminals on his desk. It was a great advance when he could open two windows on his screen and copy-paste the data. What he really wanted was a tool that facilitated the data transfer; automatically but under user control. Standalone programs were gradually replaced by subsystems that were grafted into an existing system. Communication had become the glue that joined various subsystems together.
- *Multi process.* Modern systems run on networks of distributed computers and are inherently multi-process.
- *Ownership*. Users were becoming more computer literate and demanded better control over their computer-based information and services. This put new demands on system transparency.

The answer to the new challenges was object orientation. Systems became systems of collaborating objects that have unique identity, and that encapsulates state and behavior. We first tried programming in Simula, but were stumped by its lack of persistent objects and by its typing system that forced us to break the encapsulation and think in terms of object implementations (classes). We made a preprocessor to Fortran making it somewhat object oriented. We finally ended with Smalltalk as the solution that best fitted our needs.

Smalltalk defines a virtual computer built on top of various combinations of hardware and operating systems. It is a *virtual object computer*; all programs and data have a uniform object representation. Its memory contains a large number of objects with identity encapsulating state and behavior. It is a stored program computer; programs exist as regular objects at run time and can operate on any object, including itself.

I learned Smalltalk as a visiting scientist at Xerox PARC in 1977/78 by sitting in front of the computer screen watching the objects in a debugger, changing their attributes and behavior on the fly and stepping through the program instructions. I was in intimate contact with the objects and had to bridge the gap to the problem domain in my head. Classes and subclasses were used to structure the program code, but my objects were more or less haphazardly sprawled all over the object space. I soon learned that a system consisting of some 300,000 weakly structured objects was more than my brain could master. A typical symptom was that an innocuous change in one corner of a system caused a catastrophic failure in the opposite corner.

ii: The Need for a New Generation

Even today, my programs are more like the stuff I produced in 1957 than the beautiful edifice of 1973. Thirty years of patching my 1973 software engineering discipline has not solved the problem; my brain cannot fathom my programs.

Many marketeers want us to believe that the complexity and poor quality of current software is an unavoidable consequence of the complex requirements. I claim that this is a myth, and that our problems are caused by the lack of adequate programming concepts, languages and tools. I have embarked upon a project, *BabyUML*, that shall demonstrate that it is indeed possible to write simple, transparent programs that satisfy our requirements. I hope BabyUML will enable me to write significant programs that make me feel as comfortable as I felt in 1973.

BabyUML shall produce a high level, communication centered programming discipline for a stored program object computer. It is more than a language because BabyUML shall support many languages used for describing different aspects of the systems. I have somewhat whimsically chosen the name; *Baby* because the world's first, very small stored program computer was the English *Baby. UML* because I glean many powerful constructs from UMLe.

The BabyUML success criteria are as follows:

1. *Master complexity.* Software should be transparent so that it can be mastered by the human brain.
2. *Global*. The programming discipline should scale, possibly by being recursive. It should be applicable at all levels from global architecture to local detail.
3. *Safe and secure.* Safety and security issues are getting increasingly important. Protection support must be an essential concern throughout the programming discipline.
4. *Controlled program evolution.* Requirements, design and code will evolve throughout the program lifetime. The discipline shall help maintain consistency between all three throughout the system life cycle.
5. *Support software reuse and interchange.* The discipline shall support the definition of software units that can be adapted for reuse in a variety of different contexts.

C.A.R. Hoare has given a very potent expression of our choices: *"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other is to make it so complicated that there are no obvious deficiencies."* The first relies on program transparency and readability to achieve quality programs. The second has to rely on testing. To quote Edsger Dijkstra: *"Program testing can be used to show the presence of bugs, but never to show their absence!".* The best way to avoid bugs is clearly not to put them in, in the first place. The goal of the BabyUML project is to find a discipline of programming that supports getting it right the first time. The goal of the BabyUML laboratory is to let me experiment with new methods, tools and languages that help me achieve this goal.

iii: Acknowledgements

The idea and implementation of a stored program object computer is due to Alan Kay, Dan Ingalls, Adele Goldberg and the Smalltalk team at Xerox PARC [Blue book]. UML is the combined result of a great number of people. Taken together, they have documented many concepts that are useful for modeling large systems of interacting objects.

Special thanks to Dan Ingalls for his help with the BabyUML MetaMetaclass and other Squeak constructs. Also my sincere thanks to Ragnar Normann who has helped me with database issues and who has helped me force my brain to thinking in declarative terms without immediately translating to my usual imperative code.

iv: Outline

The rest of the paper is organized as follows:

- Section A.2: *Three Generations of System Architectures* (page 8) contemplates cpu-, storage-, and communication-centered architectures. It finds that current programming languages are inadequate for communication-centered programming and suggests that we fill the gap with a new discipline of programming.
- Section A.3: *Introduction to BabyUML* (page 10) describes the BabyUML laboratory. It is based on Squeak, a dialect of Smalltalk. Its objects are embedded within a standard release of Squeak so that regular Squeak is available for the instrumentation of the BabyUML laboratory. It also reports the BabyUML meta- and metametaclasses.

The notion of an object is commonly defined as being an instance of a class. In BabyUML, an object is an independent concept being defined as an entity that has unique identity and that encapsulates state and behavior. All objects have the same characteristics when seen from the outside; they communicate by sending messages to each other. The first version of BabyUML recognizes three different realizations as described in the following three sections. The realizations are freely interchangeable in their environment, but their internals have different architectures and are supported by different tools and languages.

- Section B.1: *The Simple Object* (page 14) takes us back to first principles. We deal with the object as a concept in itself; independent of its class and other details of code. This lets us focus on the object as a partner in communication, hiding its implementation details for separate study.
- Section B.2: *The Simple Component* (page 17) discusses how we partition the object space into *components*, where a component is an object that encapsulates other objects. It can only be accessed through its *ports* that are characterized by their *provided and required interfaces*. This gives an hierarchical system composition, supporting a strategy of “divide and conquer”.
- Section C.2: *Component Merge* (page 30) discusses an extension of the concept of subclassing to the specialization of components. This idea from UML is a powerful tool for the reuse of component designs.
- Section B.3: *The Mastered Component* (page 19) has a “micro main program” called a maestro that makes the member interaction explicit, visible and manageable.
- Section B.4: *The Declarative Component* (page 22) makes the member structure explicit, visible and manageable. It is done by describing the structure declaratively, using queries to create the external views needed for different purposes.

BabyUML is still at an early stage, but I feel that some conclusions my be drawn even if a great deal of work remains to make it usable in practical programming.

- Section E.1: *Conclusion* (page 38) summarizes the work done so far and draws a conclusion.
- Section E.2: *Further work* (page 39) describes some interesting and important work that waits to be done.

“The proof of the pudding is in the eating”. The BabyUML laboratory is used to test out the ideas in practical programming:

- Appendix 2: *Component Merge* (page 46) is a quote from the UML 2.0 specification document. It defines the somewhat elusive notion of UML merge. I hope that the BabyUML version of the concept can be substantially clarified through simplification, good examples and powerful tools.

A.2: Three Generations of System Architectures

It is illuminating to compare the evolution of computer hardware architecture to the corresponding evolution of software architecture and design principles.

Figure 1: Three system architectures

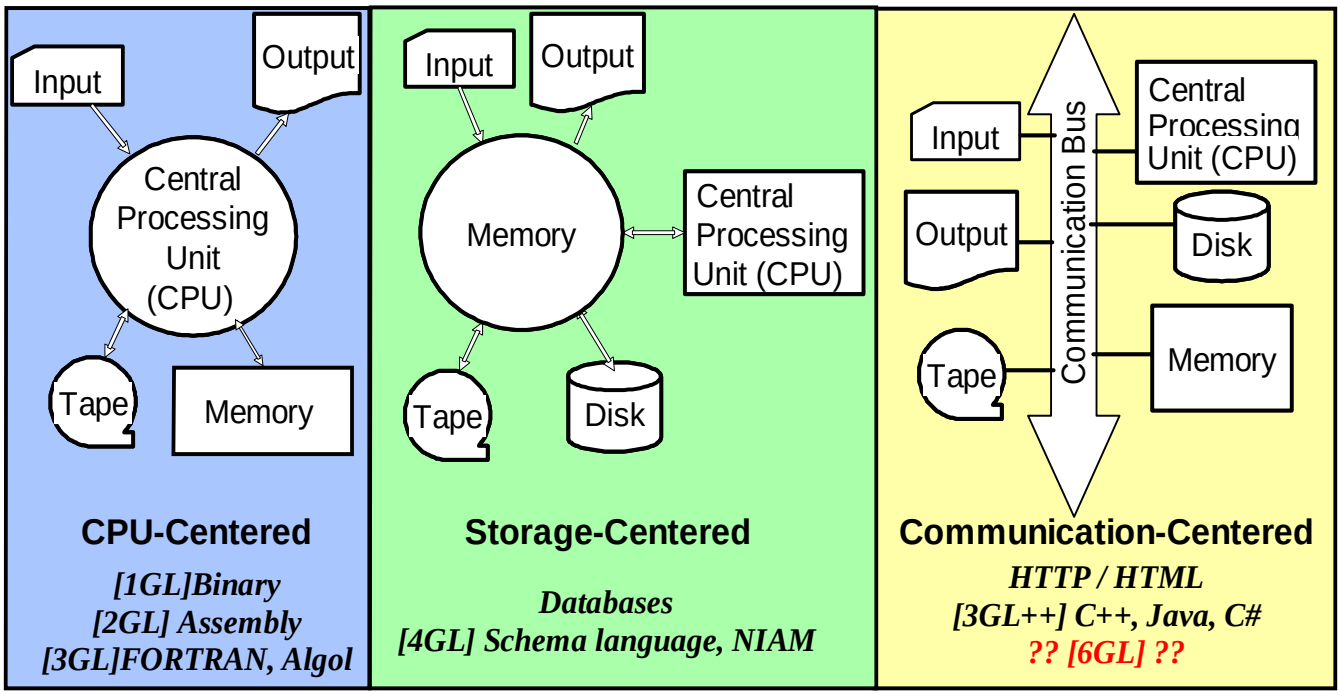


Figure 1 illustrates three generations of hardware and software architectures reflecting three different ways of thinking about data processing and programming; *CPU-Centered*, *Storage-Centered*, and *Communication_Centered* as discussed below.

i: CPU-Centered Architecture

Nusse was a binary computer; all programs and data had a uniformly binary representation. Its various units were arranged around a CPU that controlled all its operations as illustrated on the left of figure 1.

Hardware and software evolved very quickly from a primitive beginning. I wrote my first programs in binary. The next step was an assembler that enabled me to write the instructions in symbolic form. The transition to an algorithmic language was a somewhat traumatic step. The upside was that my programs were closer to the problem domain because the detailed CPU instructions were generated automatically. The downside was that I could no longer write programs that operated upon themselves. In a sense, I had lost contact with the stored program computer.

ii: Storage-Centered Architecture

For a short period in the sixties, computers were built with a “multi-port memory” in the center and various units arranged around it as illustrated in the middle of figure 1.

In 1960 , we launched our first major software development project. The product was Autokon, a CAD/CAM system that maintained its leadership in the world’s shipbuilding industry for more than 25 years. We also moved to a more powerful computer, the Swedish Facit EDB3 that for a short time was the world’s fastest. We were very fortunate with this computer because it had a “carousel” tape station consisting of a removable wheel with 64 tape spools. This made it feasible to store all information about a ship in a random access database that fitted on one carousel wheel. Various design programs were arranged around the database, accessing the data in different ways depending on the needs of the programs. This database-centered software architecture corresponds to the memory-centered hardware architecture, demoting the CPU-centered applications to be subordinated the central data store.

Applications are severely restricted in that the structuring, storage and retrieval of persistent data is the exclusive prerogative of the database system with its schemas and access mechanisms.

iii: Communication-Centered Architectures

Hardware moved very quickly to communication-centered architectures with a common “bus” where units of all kinds could be plugged in as illustrated to the right of figure 1. The architecture was soon extended to interconnect the computers themselves. The Arpanet evolved into the internet and was augmented with local intranets.

The last stage in the evolution of software architectures has gradually followed the lead from hardware. Client-server architectures such as Enterprise Java Beans (EJB) are early examples. [Web Services] is another example. Object oriented programming has entered the main stream.

Simula, Java, C++ and C# are touted as object oriented programming languages, but they are really class oriented. Java is to object orientation what Cobol is to databases because Java does not support the specification of systems of interacting objects, only their parts. We clearly need a counterpart to the discipline enforced by databases; a discipline of programming that gives us high-level control of component interconnection and interaction.

I believe a new, communication-centered discipline must subsume the CPU- and storage-centered approaches, creating a new layer of system organization on top of the other two:

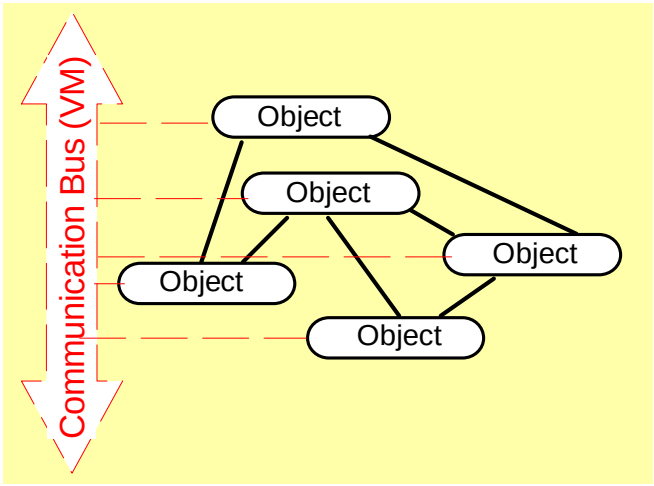
- *Communication-centered:* UML collaborations and interactions can be used to specify object communication.
- *Storage-centered:* UML class diagrams can be used to declare object structures
- *CPU-centered:* Smalltalk or Java methods can be used to specify the detailed behavior of objects.

We will unify all three architectures in the following sections.

A.3: Introduction to BabyUML

Object oriented architectures are a special case of communication-oriented architectures as illustrated in figure 2. Each object is like a virtual computer with its identity and encapsulated state and behavior. Conceptually, the objects interact with each other. In reality, the interaction is realized by a bus such as the Java or Smalltalk virtual machine (VM) or the internet.

Figure 2: Communication-centered software architecture



ComputerDistribution

Alan Kay once said that "*an operating system is what the language designers forgot to put into the language*". We could similarly claim that *UML is what the language designers forgot to put into the language*. UML is a language for specifying system models and is not directly executable. BabyUML is a discipline of programming encompassing several languages for specifying object communication, storage and computation. The UML metamodel is not directly applicable in BabyUML, but many of its constructs can be simplified and adapted to our new discipline of programming.

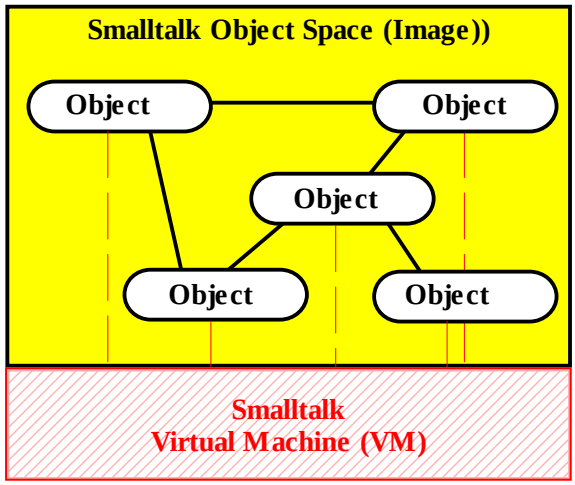
Talking and writing about a new programming discipline doesn't mean much without at least one concrete example that illustrates its feasibility and the usefulness of its proposed features. BabyUML is a laboratory for experimenting with communication-centered architectures. Most of it is still plans and other vaporware, but the skeleton of its metamodel has been implemented in the Squeak, a dialect of Smalltalk.

i: The BabyUML Stored Program Object Computer

Nusse, our computer anno 1953, was a computer were the smallest adressable unit was a *word* of 32 bits. Correspondingly, BabyUML is a computer where the smallest adressable unit is an *object* that has a unique identity and that encapsulates state and behavior.

Smalltalk [Blue book] is a virtual, stored program, object computer as illustrated in figure 3. *Object*, because its smallest unit of data is the object. (The bits of the underlying computer are hidden by the VM). All data are represented as objects; even booleans, numbers and characters. *Virtual*, because it is realized in software by the Smalltalk Virtual Machine (VM). *Stored program* because programs can be seen and manipulated as regular objects. A program can operate upon any object in the VM, including the program itself.

Figure 3: Smalltalk is a virtual, stored program, object computer



The objects are stored in the Smalltalk object space, the *image*. The VM creates a new object when told to do so by a method; returning the objectID (oop) of the new object so it can later be the receiver of messages.

Objects that cannot be reached directly or indirectly from a special root object are removed by the *garbage collector* so that their physical memory space can be reused. The Smalltalk VM also reuses the oop, but this conflicts with our basic idea of an object where the objectID is unique in space and time. We think of the Smalltalk oop as a local alias for a global, immutable object ID. The ongoing work with [DOI] could be a good starting point for a suitable standard.

The VM executes byte codes that it finds in a CompiledMethod object. A byte code can, for example, tell it to transfer a message from the current object to a receiver object. It looks up the methodDict in the receiver to find the method corresponding to the message selector. It then puts the message arguments on its stack and executes the byte codes of the new method.

The VM expects to find the format, methodDict, and superClass attributes at specific positions in the class object. The attributes are marked with orange color in figure 21 on page 28. The the remainder of the class features can be different in different class implementations.

There is a byte code for saving the image to file. It causes all objects, including the stack and the program counter, to be saved to file. This file can later be restored as an image and the execution continued at the next byte code location.

There is no way to start the execution of a new program from scratch; every execution is the continuation of some other execution. New programs are created as modifications to an ongoing execution and can be saved in a new *image file*. (Alternatively, one can save the program source code for later compilation into an image. This capability is not part of the object computer as such, but is part of a particular programming development environment).

ii: The BabyUML Laboratory is Embedded in Squeak

Smalltalk is an ideal proving ground for a new discipline of programming. The Smalltalk notions of class, method, programing language and programming tools are all realized by objects in the image. A new class is created by instantiating a *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 BabyUML can make its own variants of the default Smalltalk program objects giving them their own notions of programs, programming language and tools.

The choice of Smalltalk for implementing BabyUML was, therefore, simple. The choice of its *Squeak* dialect 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 a rapidly evolving environment with a large number of undocumented features in different states of completion. But the advantages far outweigh the objections:

- Squeak is open source; there are no obstacles to the distribution of the BabyUML 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 translated automatically to C. This means that it is feasible to modify the BabyUML VM if needed.
- But the most important argument is that there is a very active and creative community around Squeak. Many ideas and even programs that are useful to BabyUML are to be found in the Squeak databases and mailing lists.

Figure 4: The BabyUML laboratory is embedded in the Smalltalk object space

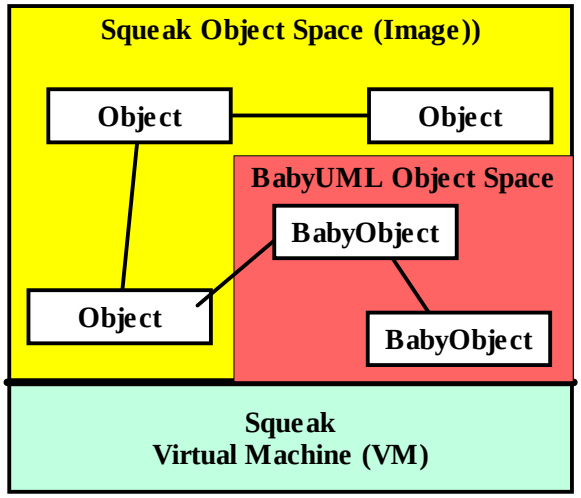
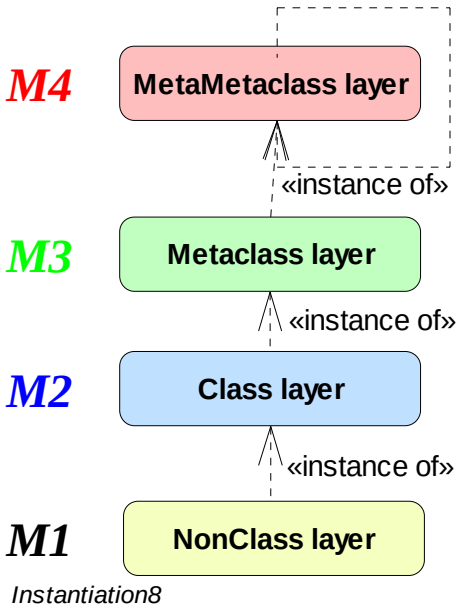


Figure 4 illustrates the BabyUML implementation. The baby objects may have their own classes, metaclasses, and metametaclasses; but they look like regular Smalltalk objects to the VM because we make them conform to its simple conventions so that they can interoperate with regular Squeak objects.

iii: BabyUML Layered Architecture

BabyUML is a *stored program object computer*. It is implemented in Squeak, a dialect of Smalltalk. All program elements including classes, metaclasses and compilers exist as tangible objects, merging program build time into run time. The Smalltalk insistence on pure object orientation invites us to shift our focus from the classes to the objects. We supersede the Smalltalk notion of programs with new high-level programming constructs gleaned from UML and transpose them to fit in the object space. The result is a set of high-level constructs that will help create clear solutions to complex problems.

Figure 5: The BabyUML Instantiation Architecture.



Instantiation8

Every object is an instance of a class. In BabyUML, this is implemented by every object having a link to its class object. Similarly, the class object 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 layered architecture is fundamental to BabyUML semantics. The layers from the concrete to the abstract are as follows:

M1 - Non-class layer: Here are the non-class objects, typically application and support objects.

M2 - Class layer: Here are the regular classes. Class objects create new instances, act as repositories for information common to these instances, and knows how to translate code from a human form to executable binary. BabyUML components are encapsulated clusters of objects. These objects have their own classes. One of them is the *component factory*; an instance is responsible for the component as a whole. (One of the experimental component factories combine a declarative description of the component members seamlessly with an imperative description of the methods that implement the component's provided operations.)

M3 - Metaclass layer: Metaclass objects create new class objects and serve as repositories for their features such as methods for instance creation and compilers. Metaclasses thus define programming languages; they can be different for different metaclasses. This ensures that BabyUML is genuinely extendable.

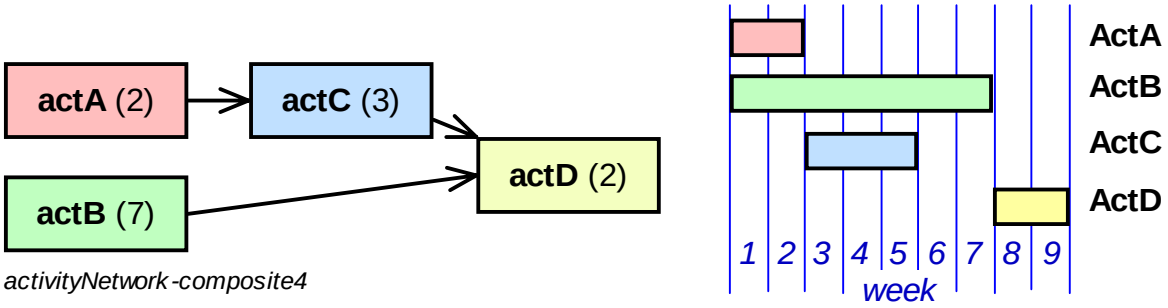
M4 - MetaMetaclass layer: There is a single object in this layer; BMetaMetaclass. This object is the mother of all objects. All objects are, directly or indirectly, instances of this class.The BMetaMetaclass object creates new metaclass objects and serves as a repository for their common features. (BMetaMetaclass, is an instance of itself so it had to be created by a somewhat tricky program.)

Note that this BabyUML layered architecture is an *instantiation* structure, a structure that is independent of the *class inheritance* structure. See figure 30 and figure 30 on page 36 for examples.

Part B: Example Objects

We will illustrate different programming disciplines through a simple example. The example is an activity network called sampleNet. The activity network is a well known approach to project planning and control. Figure 6 shows a simple network with 4 activities (tasks). The name and duration is given for each activity. We will use this example to illustrate various aspects of our proposed programming discipline.

Figure 6: The *sampleNet* activity network



Frontloading is an operation on an activity network that computes the earliest start and finish for each activity given the start time for the start activities. The result of frontloading our network from week 1 is shown as a Gantt diagram on the right of figure 6.

B.1: The Simple Object

Our systems of interacting objects will be very large. The BabyUML vision is that I shall view and code the system in different projections. Each projection shall be precise and consistent and the aggregate of all projections shall constitute the code for the running system.

We start with specifying simple objects and choose to work with them in two different projections: the *encapsulated* object and the *conceptual* object. The implementation of the object can be described in the *object descriptor* described in *"The SimpleClass"* (page 27). As an example, we show an object that represents the actD activity in sampleNet.

i: The Encapsulated Projection

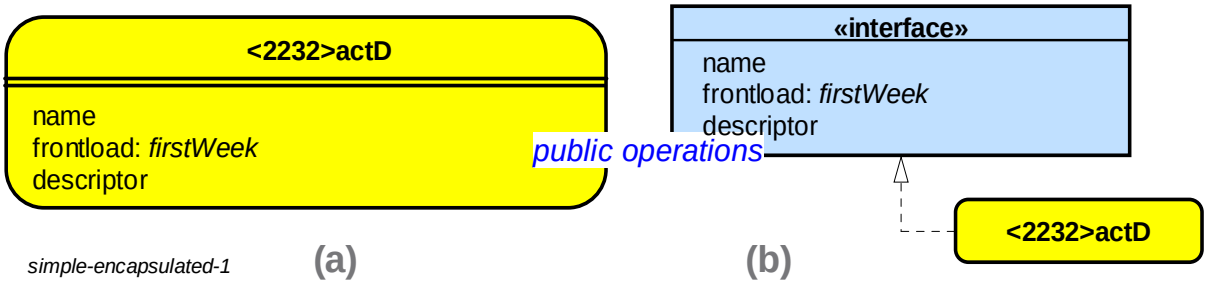
The following example of an encapsulated projection gives a “black box” view where we see the object as it appears to its environment.

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

The BabyUML object can only be accessed through its operations; object attributes and methods are invisible from outside the encapsulation. There are two mutually equivalent graphical notations for the excapsulated projection. The inline form in figure 7 (a) is useful in simple diagrams. The compact form of figure 7 (b) can be used to save screen acerage; the interface can be popped up dynamically when the mouse is over the object. The encapsulated projection is useful for “wiring diagrams” showing systems of interlinked objects.

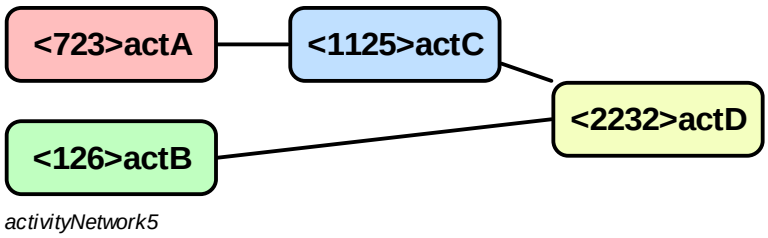
Note that the BabyUML notation for an object is a rounded rectangle. This is to distinguish it from the UML classifier. The objectID is shown in angle brackets: <2232>. Some objects have a name, this is then shown after the objectID.

Figure 7: Example encapsulated object projection.



I executed a version of the activity network example and got the objects shown in figure 8. We see that the objects are interlinked, but we do not see how this is achieved.

Figure 8: "Wired" objects implementing the *sampleNet* example.



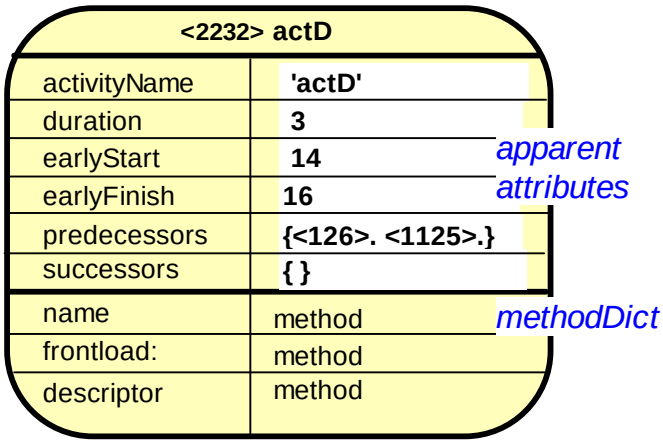
ii: The Conceptual Projection

We now look at the same object in the conceptual projection showing a "white box" view of the object. We see selected object features, but we do not see how they are realized. The conceptual projection has three compartments as shown in figure 9.

- The top compartment shows the objectID <2232> with a possible name.
- The middle compartment shows the object attributes (*instance variables*) with their names and values. A tool should show their current values as they change over time. A tool could also permit the addition and removal of attributes.
- The bottom compartment shows the operations. A tool could also show the code of the corresponding method so that it can be edited.

Issue: The conceptual projection could show apparent features, but we want the projections to *be* the program so we show implemented methods and attributes.

Figure 9: Example conceptual, white box object projection.



Object behavior is activated when the object receives a message. The message has two parts; *selector* (name) and *arguments*. The object contains a dictionary, methodDict, binding selector to method. A method is an object, it is essentially a sequence of byte codes (operations). There are byte codes for getting and setting attribute values as well as for sending messages to specified objects. Object orientation gets much of its power from *polymorphism*. In BabyUML, this means that methods are local to the object so that different objects receiving messages with identical selectors may handle them differently.

Note that these projections show the *object*. The actual arrangement of classes and superclasses is irrelevant from the object's point of view. Section C.1: *The SimpleClass* (page 27) describes the implementation of our example object.

B.2: The Simple Component

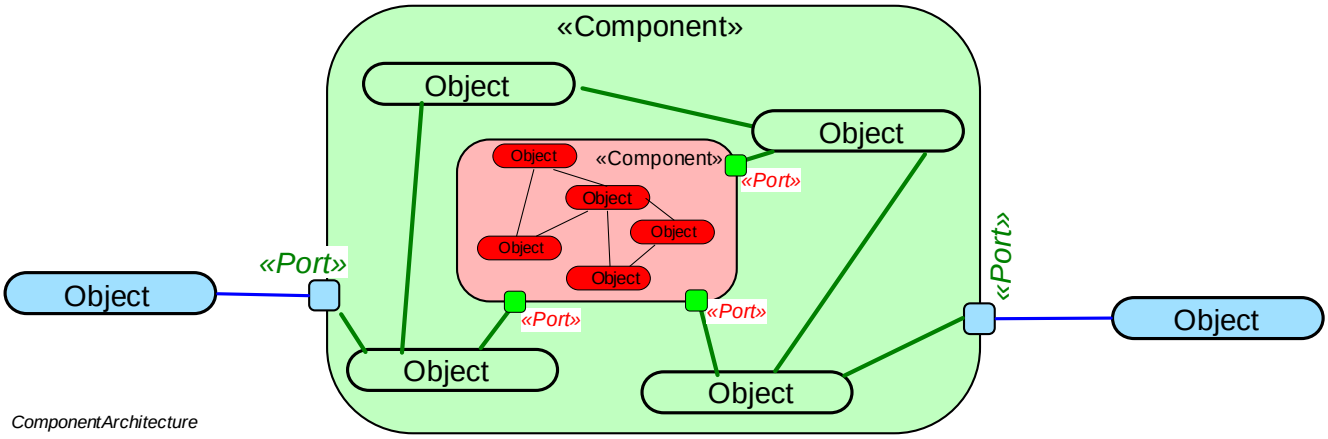
The object-centered architecture of figure 2 on page 10 is both flexible and powerful. Systems are easy to master as long as they don't get too large and too complex. But I have for years been working with systems containing several hundred thousand objects. They exist in a flat object space and I need to partition them in order to master them.

i: A Component Encapsulates Objects

An object is encapsulated; its internal realization is invisible from its outside. [UML] has the notion of a *component* that is a kind of class. In BabyUML, a component is an object that encapsulates other objects making them invisible from the outside. The notion is recursive; I can organize my several hundred thousand objects in a tree structure so that I deal with a manageable number at each level. In figure 10, I show a system as a component within a component. Both components are expanded so that we can see their internal objects.

The upside is that I may attain mastery of my objects. The downside is that many of my current object structures will be outlawed. I believe that the restriction is essential to the mastering of complexity because it will help me avoid the kind of errors where a change in one corner of a system causes a failure in the opposite corner.

Figure 10: A Component is an object that encapsulates other objects



A *port* defines a distinct interaction point between a component and its environment. The BabyUML *port* is an object that passes messages between the component's internal objects and the objects in its environment. In figure 10, the objects in the outer, blue world can only access the middle, green world through the blue ports. Conversely, the objects of the middle, green world can only access the outer world through the ports. The links between a component and its environment are applications of a facade-like pattern [GOF].

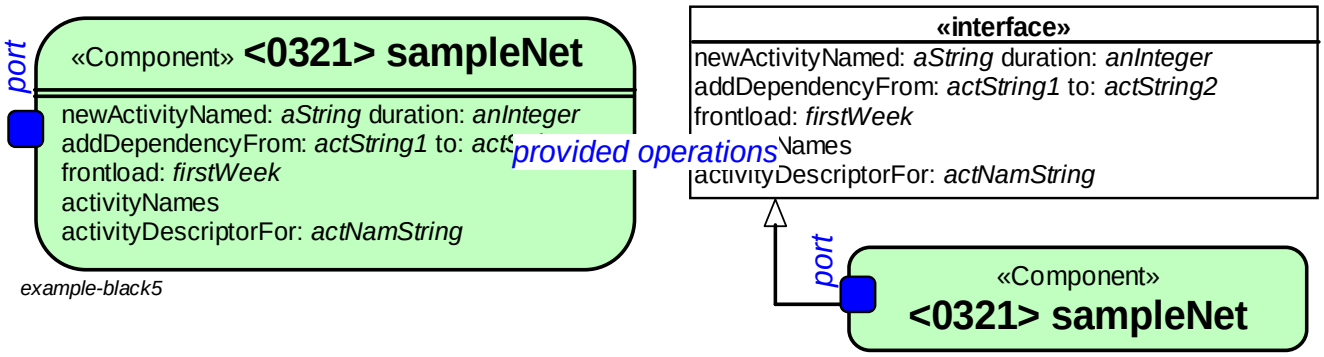
A final comment about high-level constructs:

> *There can be no leverage without rigidity. Fortran facilitates the programming of formula evaluation while it prohibits self-modifying programs. Structured programming facilitates simple program structures while it prohibits many intricate execution patterns. My own transition to structured programming was very traumatic; I struggled for almost half a year to create subroutines without GOTOs. But then my mind suddenly clicked, and I now cannot understand why GOTOs were so important.*
>
> *I suspect that a transition to communication-centered programming can be equally traumatic. The way I think about programs will have to change, many dear constructs will no longer be permissible, and many work patterns will no longer be viable. BabyUML is a laboratory for experimenting with the trade-off between flexibility and leverage. The laboratory is described in ????*section A.3 on page 10*????.*

ii: Encapsulated Projection of a Simple Component

The BabyUML SimpleComponent is a component that has a provided interface offered through its single provided port. Seen from its environment, a SimpleComponent looks like a regular object with objectID and provided operations as can be seen by comparing the object in figure 7 on page 15 and the component of figure 11.

Figure 11: Two encapsulated projections of the sampleNet component

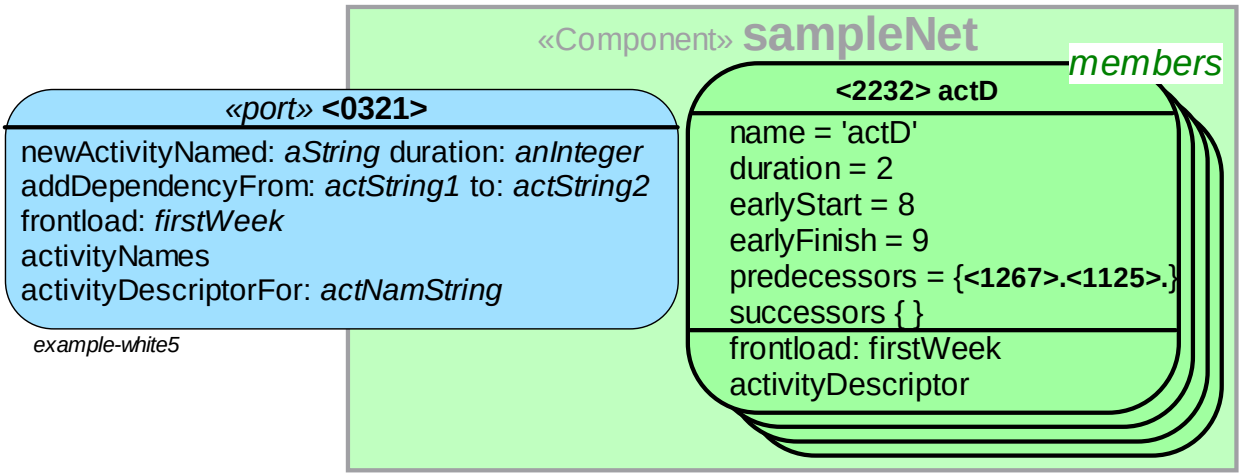


The **sampleNet** component is an object that realizes a particular activity network. Its members appear to be a number of activities, they are not visible from outside the component. The component can only be accessed through its *provided operations* as shown. The two first operations are used to build the network, the middle to perform the frontloading operation, and the last two are query operations returning textual descriptions of the network.

iii: A Conceptual Projection of the Example Component

Objects and components are interchangeable in their environment, and sampleNet could be realized by either. The difference becomes apparent if we go inside the encapsulation to get a conceptual projection of the internals. The simple object has features as shown in figure 9 on page 15, while the simple component is a somewhat more complex structure with port and members as illustrated in figure 12.

Figure 12: Conceptual projection of the sampleNet component.



The <0321> port object is visible in the component environment; its objectID is the component's ID. Hidden within the component are the members, the activity objects. The port and the members are shown as conceptual objects.

B.3: The Mastered Component

The simple component frontloading code given in *Excerpts of the sampleNet code* on page 34 looks clean and simple at a first glance, but it illustrates one reason why I have lost control over my software when the object interaction pattern gets complex. The difficulty is that the very meaning of *frontloading* is implicit and hidden in the details.

Figure 13: Simple, distributed frontloading operation

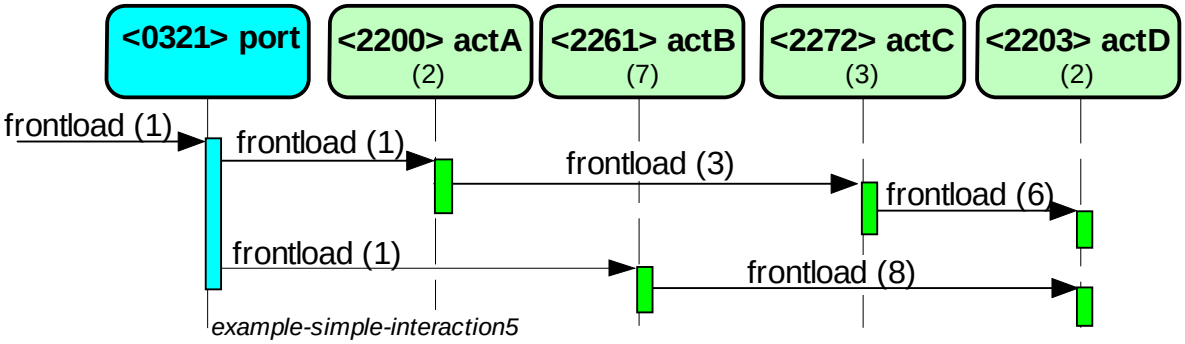
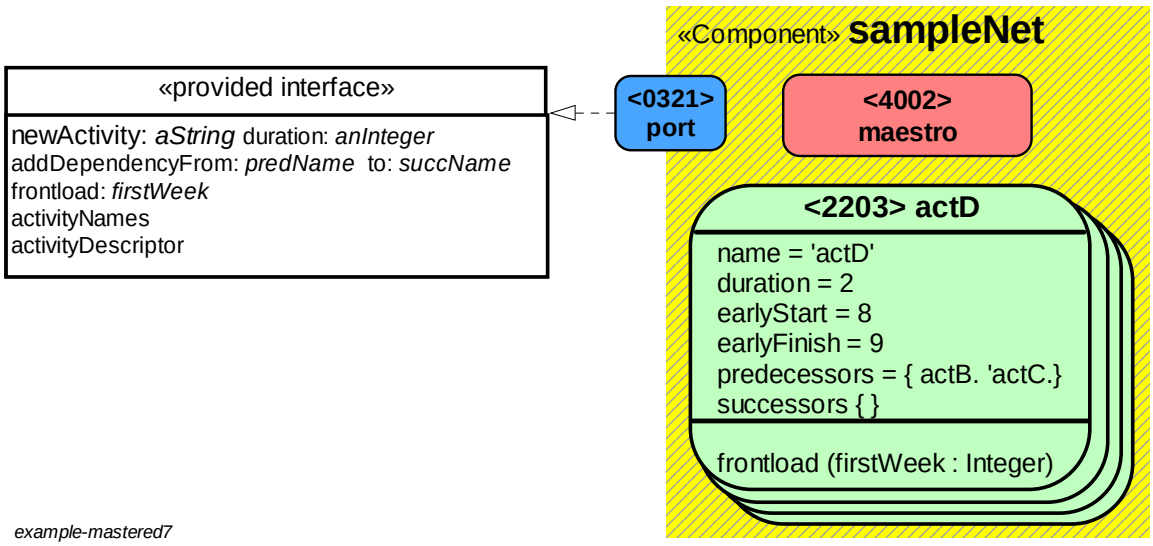


Figure 13 shows that the frontloading process for this simple example is realized by six activations. Consider a realistic situation with a complex structure and many different activity classes. I would need to read and remember a considerable body of code to ascertain what happens at run time. I would have to read the detailed code to ascertain if and how the predecessors and successors variables form the true network. I also have to check the detailed implementation of the forntloading methods to understand the interaction algorithm. There is always a danger that a future maintainer in a hurry misses important aspects and fixes a particular problem in a way that plays havoc with the original intention.

I suggest that a solution is to refactor the component internals to centralize the essentials and distribute the details.

Figure 14: The sampleNet mastered component with its objects.



We have seen how we can partition an object space into a hierarchy of components so that each component only contains a manageable number of parts. We now go one step further and see how the component as a whole can be responsible for its provided operations. The result is the *mastero-controlled component* as is illustrated in figure 14. Its objects are as follows:

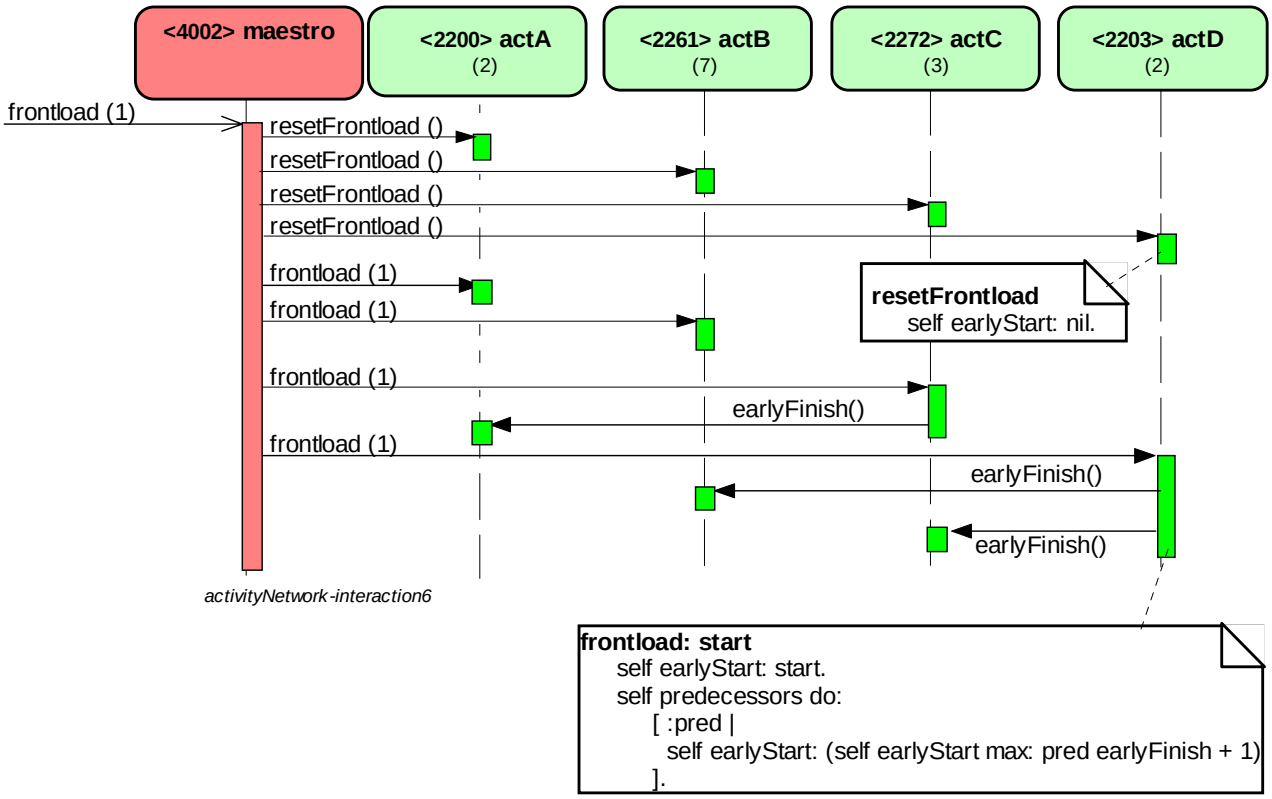
- **port<0321>** Objects in the environment still access the component trough its port.
- **maestro<4002>**. We think of the component as an orchestra. The members are the instrumentalists; they do most of the work. A very small ensemble can do without a conductor; but a full blown orchestra must have a *maestro* who is responsible for the performance as a whole; who selects the

players, assigns their roles and synchronizes their performance. Correspondingly, the maestro object is an integral constituent of the component, being responsible for the whole and for directing component interactions.

- **activities**. There are any number of activity objects. They are demoted from their prominence in the earlier, simple implementation since they are no longer responsible for the overall object interaction patterns.

The maestro receives the provided operations through the port <0321>. Each operation is implemented as an interaction that is triggered from the port, orchestrated by the maestro<4002>, and performed by the members as shown in the sequence diagram of figure 15.

Figure 15: The maestro controls component interaction



The small rectangles on the lifelines represent method activations. We show two of them as comments in the diagram[1]; a tool could show a selected method in a separate pane where it could be edited.

The activity computes its earlyStart under the assumption that its predecessors know their earlyFinish times. It is the maestro's responsibility to ensure that this invariant is satisfied. The maestro, therefore, sends the frontload-message to selected activities that are either start activities with no predecessors or activities with predecessors that have already been frontloaded:

```
Maestro>>frontload: firstWeek
        | frontActs |
        activities do: [:act | act resetFrontload]. " Set earlyStart to nil. "
        [(      frontActs := activities select:
                        [:act |
                                act earlyStart isNil
                                and: [act predecessors allSatisfy: [:pred | pred earlyStart notNil]]]
         ) notEmpty]
        whileTrue:
                [frontActs do: [:act | act frontload: firstWeek] ]
```

1. We here switch to Smalltalk because it is more compact and because it is the basic language of BabyUML.

We have now made the member interaction explicit and visible in the maestro code, while the details are still delegated to the activity parts as illustrated in the sequence diagram in figure 15.

B.4: The Declarative Component

In section B.2 on page 17, I partitioned the object space into an hierarchy of encapsulated components. I also showed how my traditional programming style effectively hid system structure and behavior inside the individual member objects. In section B.3 on page 19, I moved responsibility for the overall interaction from being implicit in the component member objects to becoming explicit in the maestro. I will now move responsibility for the object structure from being implicit in the member objects to becoming explicit in the component itself.

Figure 16: UML class diagram

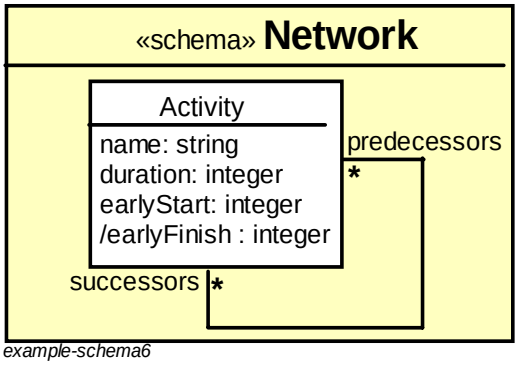


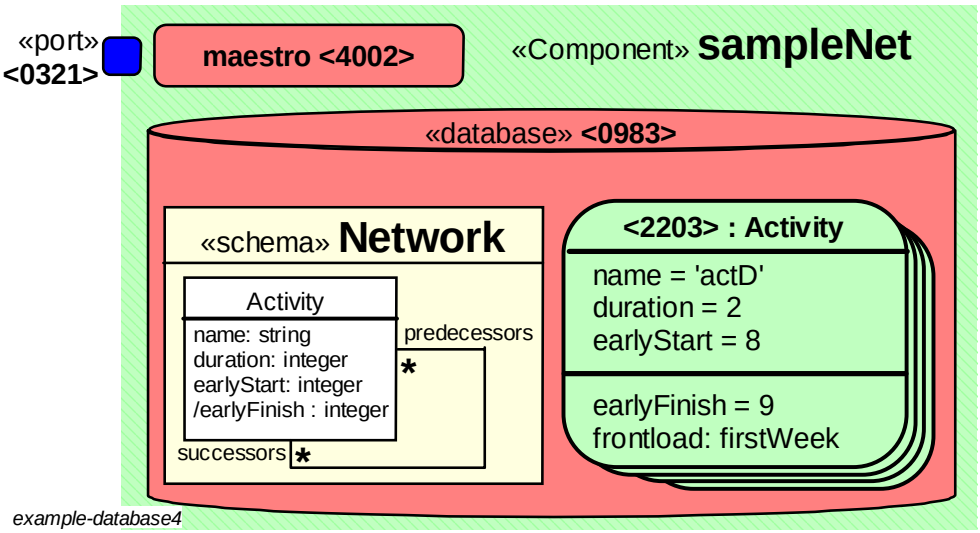
Figure 16 describes the sampleNet activities in a UML class diagram. This diagram can be interpreted as defining the Activity class; a tool can generate the class definition code such as the Smalltalk definition given on page 34. This use of the class diagram is powerful, but it also has its limitations. The code does not exhibit the important information that the predecessor/successor relationship is binary; an activity shall always be included in the successors of its predecessors. Equally important is that an interested person can only extract this information by carefully reading the code details. Inspection of a network instance can give a hint, but there can always be hidden exceptions motivated by special cases or caused by bugs .

We therefore introduce a new kind of component where the class diagram is interpreted as a declarative schema for a micro *database* that contains the component member objects. The diagram will be part of the component code; a reader can thus trust that this is indeed what is meant by an activity network in this component. Referential integrity is guaranteed and, as an added benefit, member objects can be persistent objects stored in a private database local to a component instance.

i: Storage-Centered: A Micro “Database” Holding the Members of the Component

Figure 17 shows a component descriptor for our sampleNet with the component members stored in a private database.

Figure 17: Component descriptor for a declarative component.



The component schema can be expressed in any schema language such as SQL, OSQL, [ODMG]/ODL, [UML]/OCL,... As an example, we express the schema of figure 16 in the Object Definition Language of [ODMG]. We have simplified the activity objects by representing the structure information in separate Dependency objects:

```
class Activity
    (extent activities)
    { attribute string name;
      attribute short duration;
      attribute short earlyStart;
      attribute short earlyFinish;
    };
class Dependency
    (extent dependencies)
    { attribute Activity pred
      attribute Activity succ
    };
```

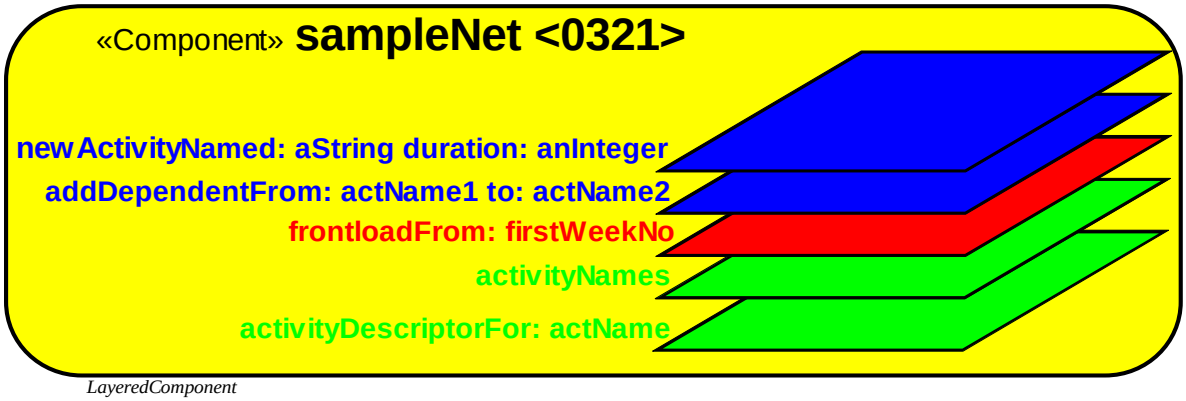
The choice of schema language is not important here because even the schema language can be local to the component. We expect many programmers will prefer to use their regular OO language; the essence being to centralize the structure information. We here encode the schema in Smalltalk to keep our example as simple as possible:

```
Object subclass: #Base
    instanceVariableNames: ‘activities associations’. ..
Object subclass: #Activity
    instanceVariableNames: ‘name duration earlyStart’...
Activity>>earlyFinish
    ^^self earlyStart + self duration - 1.
Object subclass: #Association
    instanceVariableNames: ‘pred succ.’
```

ii: Communication-Centered: Maestro-controlled interaction

We are now ready to realize the component's provided operations. The idea is to separate the component into layers, each layer specifying how the component shall implement one or more of the provided operations as illustrated in figure 18.

Figure 18: Layered component description.



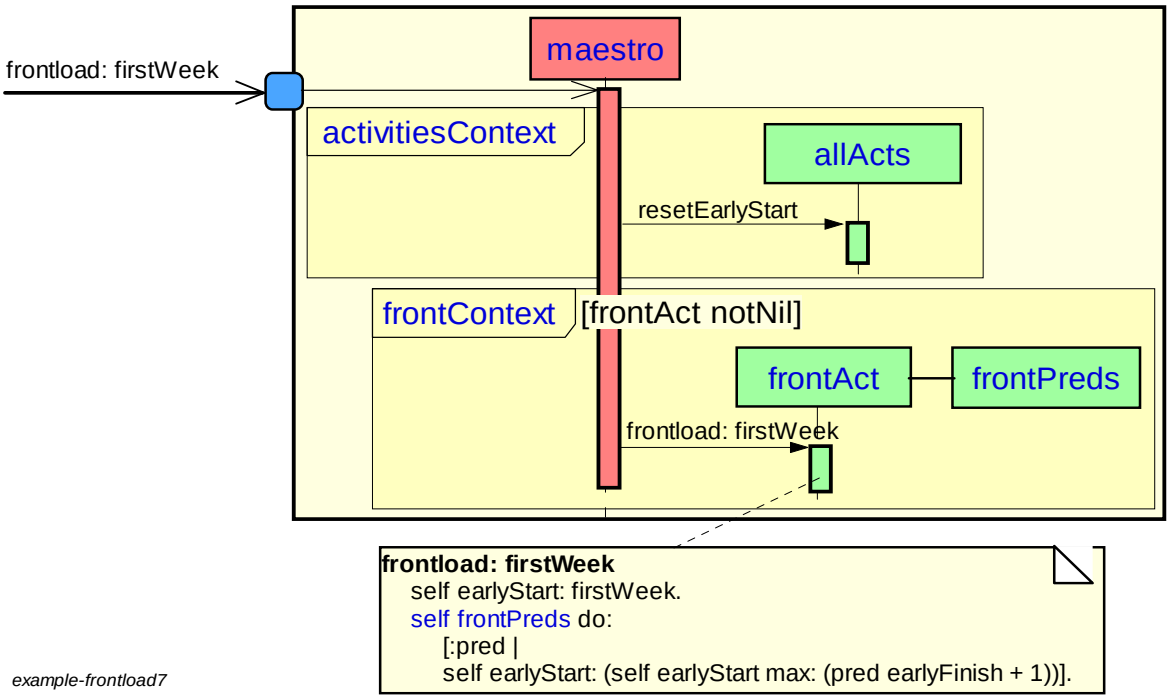
LayeredComponent

The *collaboration* defined in [UML] can be adapted to programming the provided operations:

> *A collaboration describes a structure of collaborating objects (roles), each performing a specialized function, which collectively accomplish some desired functionality. Its primary purpose is to explain how a system works and, therefore, it typically only incorporates the relevant aspects of the objects. Thus, details, such as the identity or class of the actual participating objects are suppressed.*
>
> *A collaboration specifies a projection (or view) of a set of cooperating members. It describes the required links between members that play the roles of the collaboration, as well as their features. Several collaborations may describe different projections of the same objects.*

We can program each provided operation in separate collaboration with its associated roles and interactions. The frontloading operation is most interesting, and we discuss it here.

Figure 19: Frontloading sequence diagram.



example-frontload7

As with the MasteredComponent, the provided operations are realized by the component maestro object. The maestro method for frontloading is coded in a sequence diagram in figure 19. The method consists of two nested fragments called activitiesContext and frontContext.

A fragment is executed in a context, which is a dictionary (namespace) where role names are dynamically mapped to component members. The name of the context itself is the name of a query to the database.

The appropriate language mechanisms for mapping role name to member in the code is still under study. The contexts should probably be on the stack to make them private to the interactions.

We need to perform three steps to realize the frontloading operation. We start by assigning component member objects to the roles:

- The maestro role is played by a special object within the component.
- The frontAct role is played by an unplanned activity object with planned predecessors.
- The frontPreds role is played by the predecessors of the frontAct.

iii: Mapping roles to members

This mapping can be expressed as database queries, for example in the Object Query Language of [ODMG]:

```
activities
```

is already defined as the extension of Activity.

The role named frontActivity is any one of the this set:

```
define query frontActivities as
    select act
    from activities
    where act.earlyStart.isNil
    and
        (   select succ
            from dependencies
            where pred = act
            and pred.earlyStart.notNil)
        ) isEmpty
```

And the predecessors:

```
define predecessors (Activity act) as
    select pred
    from dependencies
    where succ = act
```

In Squeak, we execute the interaction in context:

```
Base>>inActivitiesContextDo: aBlock
    allActivities := activities.
    ^aBlock value.
```

```
Base>>inFrontContextRepeat: aBlock
    | frontActs |
        [frontActs := activities select:
            [:act |
            act earlyStart isNil
            and:
                [(associations select: [:assoc | assoc succ = act and: [assoc pred earlyStart isNil]]
                 ) isEmpty]
            ].
        frontActs notEmpty
    ]
    whileTrue:
    [   frontAct := frontActs anyOne.
        frontPreds := (self predecessorsOf: frontAct) asArray.
        aBlock value
    ].
```

iv: Member interaction

We realize the frontloading operation as a maestro method. Figure 19 shows a UML sequence diagram realizing the operation. The sequence diagram is one of the very powerful graphical languages for behavior provided by [UML] and the maestro code for frontloading can be compiled directly from this diagram.

Many programmers may prefer to code it textually. The following code is meant to be equivalent to the diagram:

```
Maestro>>frontload: firstWeek
    base inActivitiesContextDo:
        [self allActivities do: [:act | act resetFrontload]
        ].
    base inFrontContextRepeat:
        [ self frontAct frontload: firstWeek.
        ].
```

v: CPU-Centered programming of the details

We wrote the default details in the context of the interaction in figure 19. This code can be specialized in different activity classes, but the programmer is severely constrained because the only visible members are the ones visible in the context. This restricts the code writer's scope for ingenuity and the code reader's scope for confusion.

```
Activity>>frontload: firstWeek
    self earlyStart: firstWeek.
    self frontPreds do: [:pred | self earlyStart: (self earlyStart max: (pred earlyFinish + 1))].
```

Part C: The Classes

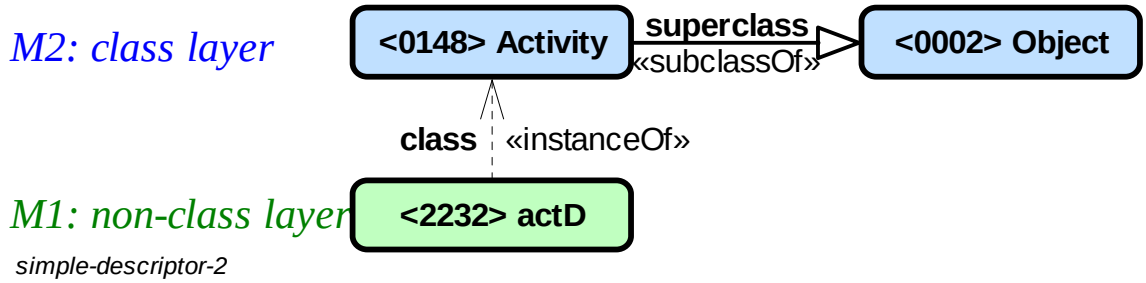
C.1: The SimpleClass

We gave an example of a simple object in section B.1: *The Simple Object* (page 14) and showed two projections. The external showed the object as a black box, while the conceptual showed the object with its features.

i: The black box Object Descriptor

The conceptual, white box object depicted in figure 7 on page 15 would be very inefficient if all the information were stored in every object because there would be a great deal of duplication. The implementation actually only stores the attribute values in an *instance* as shown with bold text on a white background in the figure. The rest of the information is delegated to its *class* and *superclass* objects as illustrated in figure 20. We coin the term *Object Descriptor* to denote this realization of an object consisting of an instance, its class object and the superclass objects.

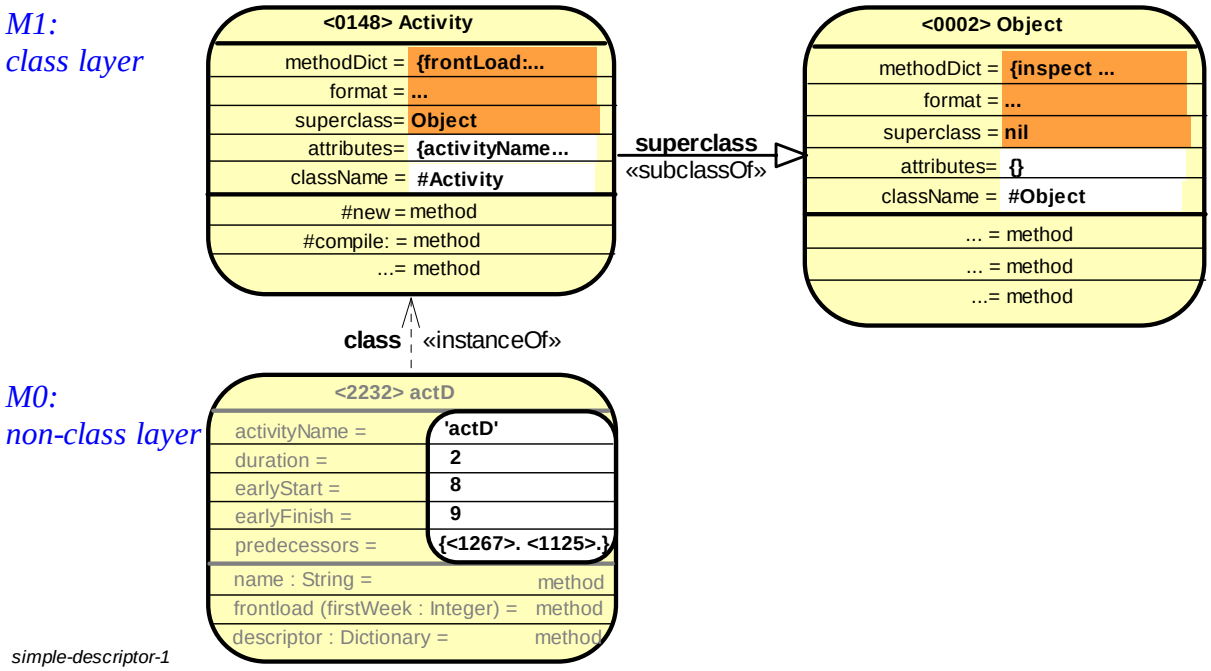
Figure 20: The Object Descriptor consists of several objects



ii: The object implementation

There are two important kinds of relationships in figure 20; the «instanceOf» and the «subclassOf» relationships. An object descriptor consists of exactly one «instanceOf» followed by any number of «subclassOf». The objects of figure 20 are expanded into conceptual objects of the object descriptor in figure 21.

Figure 21: Example object conceptual descriptor



The objects are as follows:

- The instance itself has objectID=<2232> and stores the values 'actD', 3, 14, 16, {<1267>. <1125>.}.
- Every object is an instance of a class and has a link to it. Our object 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 here nil, thus terminating the superclass chain.
- The names of the object's attributes are the union of the attributes attributes of the class and all its superclasses.
- The object's methods are a union of the methods defined in the methodDict attribute of its class and all its superclasses.
- Note that the frontload: - method is stored in the class <0148>Activity. The inspect-method is stored in the class <0002>Object. All of them are visible to the collaborators of the <2232> object as bona fide operations on that object.
- Also note that the class object <0148> has its own message interface to its own methods. In this particular implementation, it responds to the message new, telling it to create a new instance of itself. It also responds to compile: *sourceText*. The corresponding method is a compiler that translates the sourceText into a method with its byte codes and installs the method into the methodDict for later execution by the instances of this class, e.g., <2232>. In BabyUML, there will be compilers that translate various code projections into executable programs.

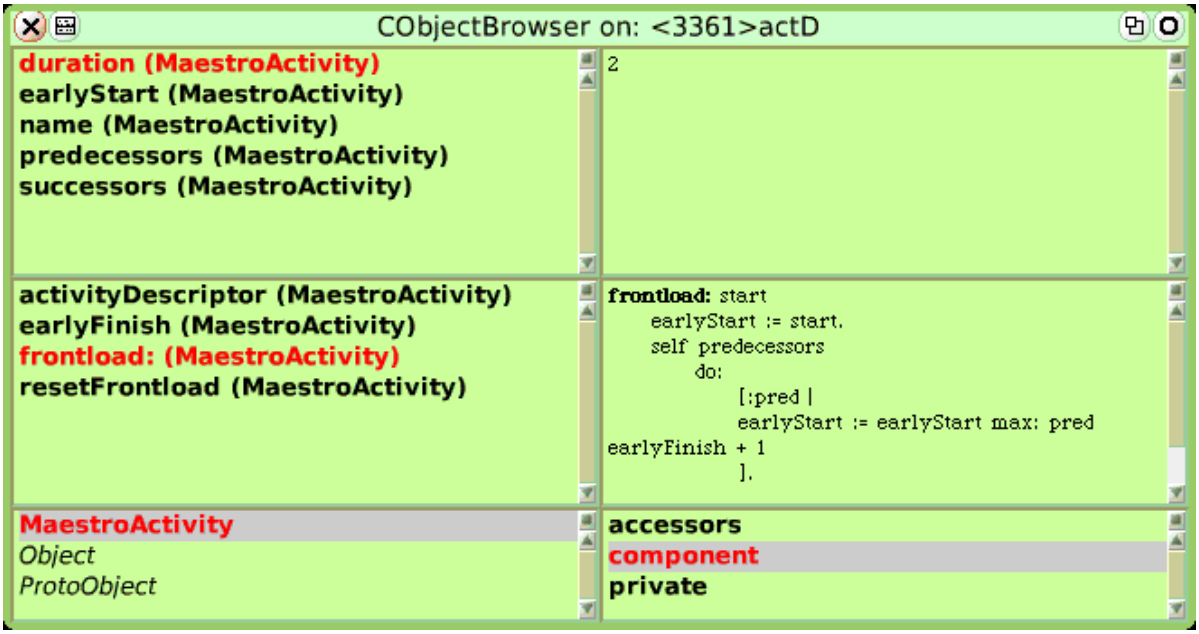
Every object is an instance of some class. The class of a class object is called a *metaclass*. It is a rich source of confusion that the features of the <2232>actD object is given by the descriptor objects shown in figure 21, while the features of the <0148>Activity class object is given by a different descriptor. This is discussed in more detail in section D.1: *The MetaSimpleclass* (page 35)

iii: Browsing the Object Descriptor

A person exploring the object space or editing the object methods can do so with the *ObjectBrowser* shown in figure 22. The bottom row contains two multi-select lists that are used to filter the views on class/ superclass and interface respectively. The middle row shows the operations with the corresponding methods, and the top row shows object attributes and their current values.

The browser supports the editing of the class features. Superclasses appear as read only because it is felt that the programmer should not change the superclasses without seeing the consequences.

Figure 22: An Object Browser IDE accesses the object descriptor

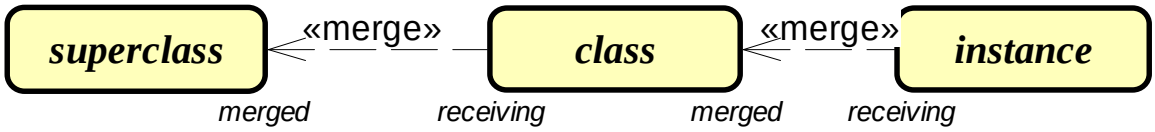


Note: The above screen dump shows an objectID that is different from other actD examples because it was captured from a different execution of the sampleNet test.

C.2: Component Merge

The merge relationship is an innovative and powerful construct that was introduced in [UML]. An extract of the original [UML] definition is given in appendix2 on page46 where we see that the original is a construct for merging model packages. In BabyUML, we explore a similar construct for merging components as exemplified in figure 23.

Figure 23: The component *merge* operation.



Smalltalk-merge1

A *component merge* is an operation on two components that makes the contents of the two components appear combined. It is very similar to generalization in the sense that the *receiving* component conceptually adds the characteristics of the *merged* component to its own characteristics resulting in a *conceptual* element that combines the characteristics of both.

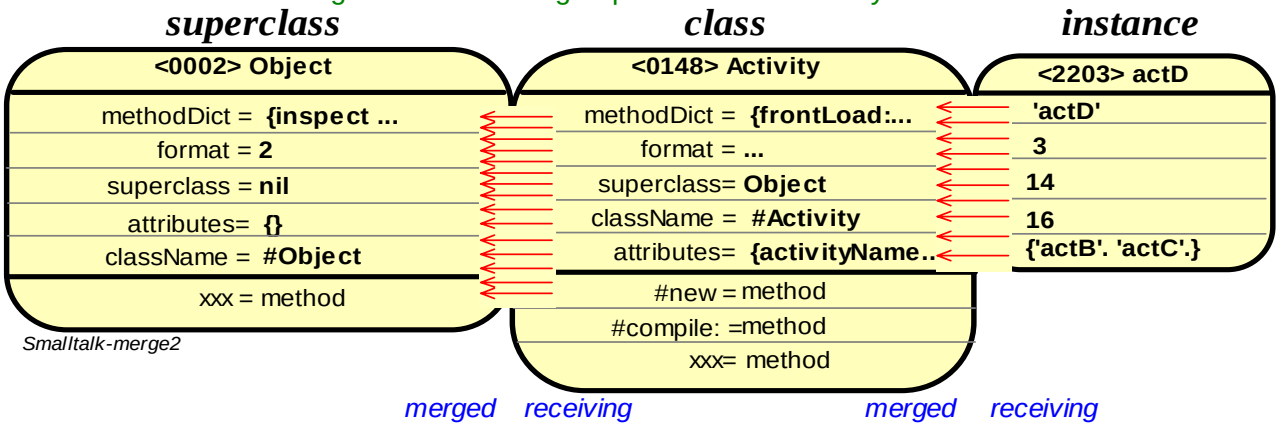
i: The Object Descriptor described as Merged Components

A BabyUML component is encapsulated and can only be accessed through its provided ports. The merge operation extends this rule so that a *merged* component can appear as being an integral part of a *receiving* component; the result being a flattened description called the *conceptual* component.

Figure 23 illustrates how the object descriptor of figure 21 on page 28 can be defined in terms of the merge operation. We see that the conceptual, white box object in figure 9 on page 15 is the result of two merge operations. The first flatten the superclass into the class. The second makes the combination conceptually becoming part of the instance. An intuitive illustration is given in figure 24.

In terms of program semantics, there is no difference between a black box program descriptor with explicit component merges, and a white box conceptual program where all the merges have been performed. The black box views of figure 23 and figure 24 are, therefore, semantically identical to the white box view of figure 9 on page 15.

Figure 24: The merge operator as a one way mirror.



Smalltalk-merge2

The merge operation is intuitively like a one way mirror between the merged and receiving components:

- The elements of the merged component appear as integral parts of the receiving component.
- The elements of the receiving component are invisible from the merged component.
- Additionally, we may find it useful to require that the state of the merged elements shall not be changed from the receiving component.

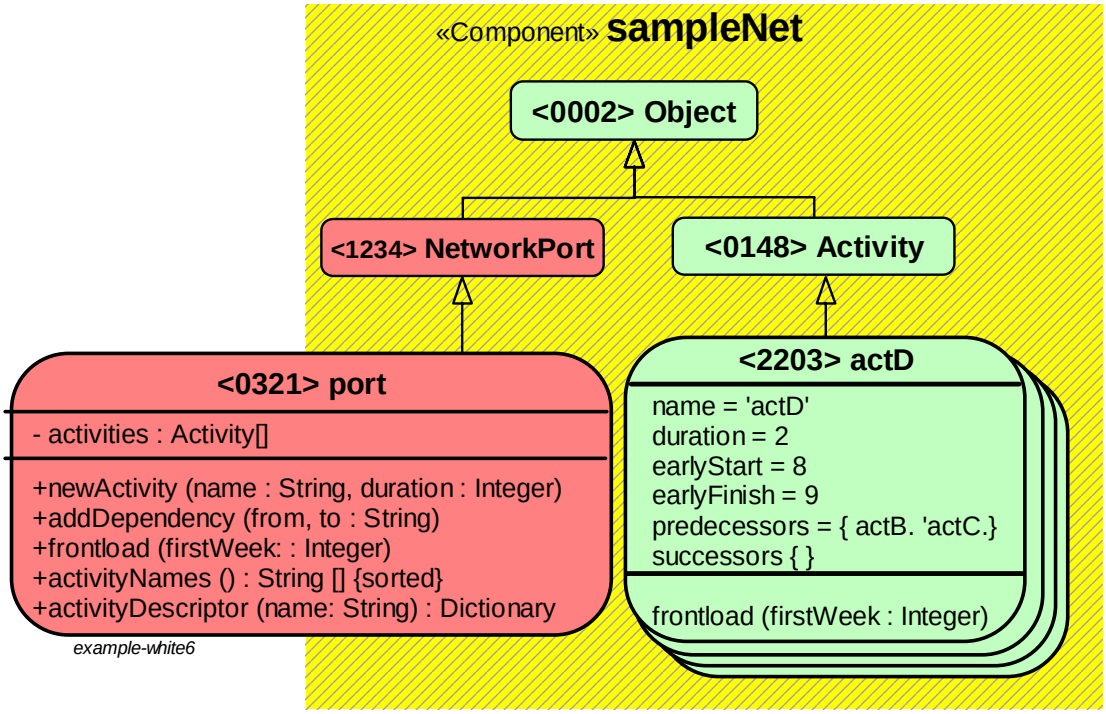
The white and black box views are all projections of the same program and the programming environment continuously synchronize them with the virtual machine by applying the appropriate transformations. In this example, we have the «superclass» and «instance of» relations, but other relations can be added as needed. Some examples that will be studied in the future:

- *Traits* dissociates methods from the classes and define fine grained relations between them. [Schärli-03] presents traits as a simple compositional model for structuring object-oriented programs. A trait is essentially a group of pure methods that serves as a building block for classes and is a primitive unit of code reuse. In this model, classes are composed from a set of traits by specifying glue code that connects the traits together and accesses the necessary state. Methods defined in different traits can be merged into a flattened, conceptual trait using the traits glue specifications.
- The OOram *role model synthesis* [OORAM] is a sophisticated relationship between UML collaborations where roles in different collaborations are constrained to be played by the same object. Collaborations can be projections of BabyUML programs; they could be merged through the synthesis relationship.

ii: The sampleNet Specified as Merged Components

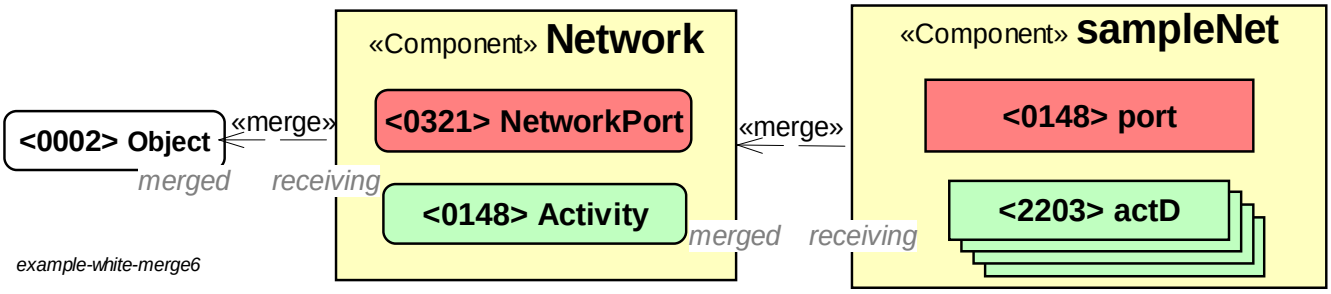
The sampleNet conceptual projection of figure 12 on page 18 can be realized if the NetworkPort and Activity classes are kept outside the component and accessed through required interfaces. This seems somewhat cumbersome, and we prefer to keep them inside the component. Figure 25 shows a conceptual view of the network component with its classes.

Figure 25: Conceptual network component.



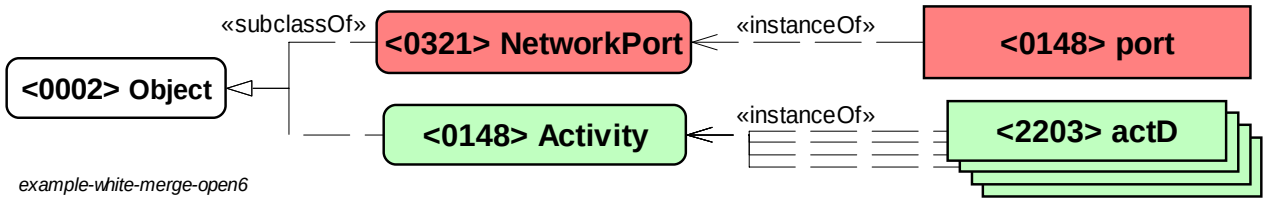
In figure 26, we show the merged and receiving components that together make up the conceptual component in figure 25.

Figure 26: Merged and receiving network components



The relationships that are implied in figure 26 are shown explicitly in figure 27. We expect that the first shall be all that is seen by the programmer; the detailed relationships should be managed automatically by the programming tools.

Figure 27: The individual relationships between the objects of figure 26.



The two last chapters have shown many different ways for viewing a program in a stored program virtual object computer. They are all illusions, the virtual computer deals with simple objects, classes and methods. Our programming discipline is an abstraction buildt on top of this; the diagrams are program source code in our interactive development environment.

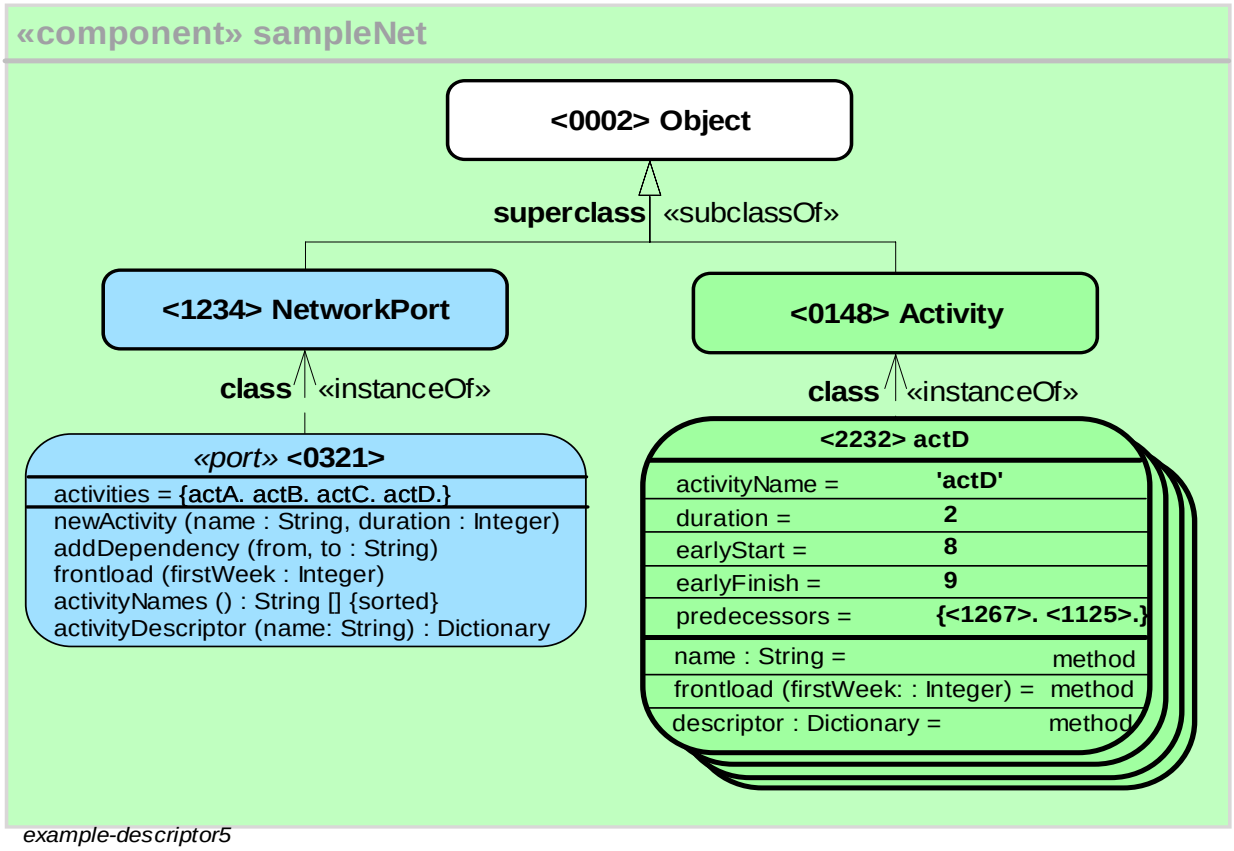
C.3: The Simple Component

i: The Simple Component Descriptor

The SimpleComponent is indeed simple. Figure 28 suggests that the sampleNet descriptor consists of eight objects:

- Four activity instances holding the attribute values for the four example activities.
- One Activity class object holding some features common to the activity instances.
- One port instance forming the bridge from the environment and the activities. The port receives messages from the environment and passes them on to the appropriate activity objects. The identity of the port is the identity of the component as a whole.
- One NetworkPort class holding some of the port features.
- One Object class object holding features common to all instances.

Figure 28: sampleNet Component Descriptor.



example-descriptor5

The component as a whole does not exist as an object, so we have shown it rectangular and grayed out. The component is not enforced by the programming environment, but is implemented by programmer discipline. We will later see more powerful components where the discipline is enforced by a component object.

ii: Implementation of the network component class

xxxx

iii: Excerpts of the sampleNet code

I first coded the sampleNet component of figure 12 on page 18 in my usual style, distributing state and behavior in a structure of interacting activity objects: [1]

```
AbstractActivity subclass: #SimpleActivity
        instanceVariableNames: 'predecessors successors counter'
    poolDictionaries: ''
```

The activity frontload operation:

```
SimpleActivity>>frontload: start
        self earlyStart
                ifNil: [self counter: 0. self earlyStart: start]
                ifNotNil: [self earlyStart: (start max: self earlyStart)].
        self counter: self counter+1) >= self predecessors size
                ifTrue: [self successors do: [:succ | succ frontload: self earlyFinish + 1] ].
Object subclass: #SimpleNetworkPort
        instanceVariableNames: 'activities'
        poolDictionaries: ''
SimpleNetworkPort>>frontload: firstWeek
        activities do: [:act | act resetFrontload].
        (activities select: [:act | act predecessors isEmpty])
                do: [:act | act frontload: firstWeek]
```

1. Ie here switch to Smalltalk because it is more compact and because it is the basic language of BabyUML.

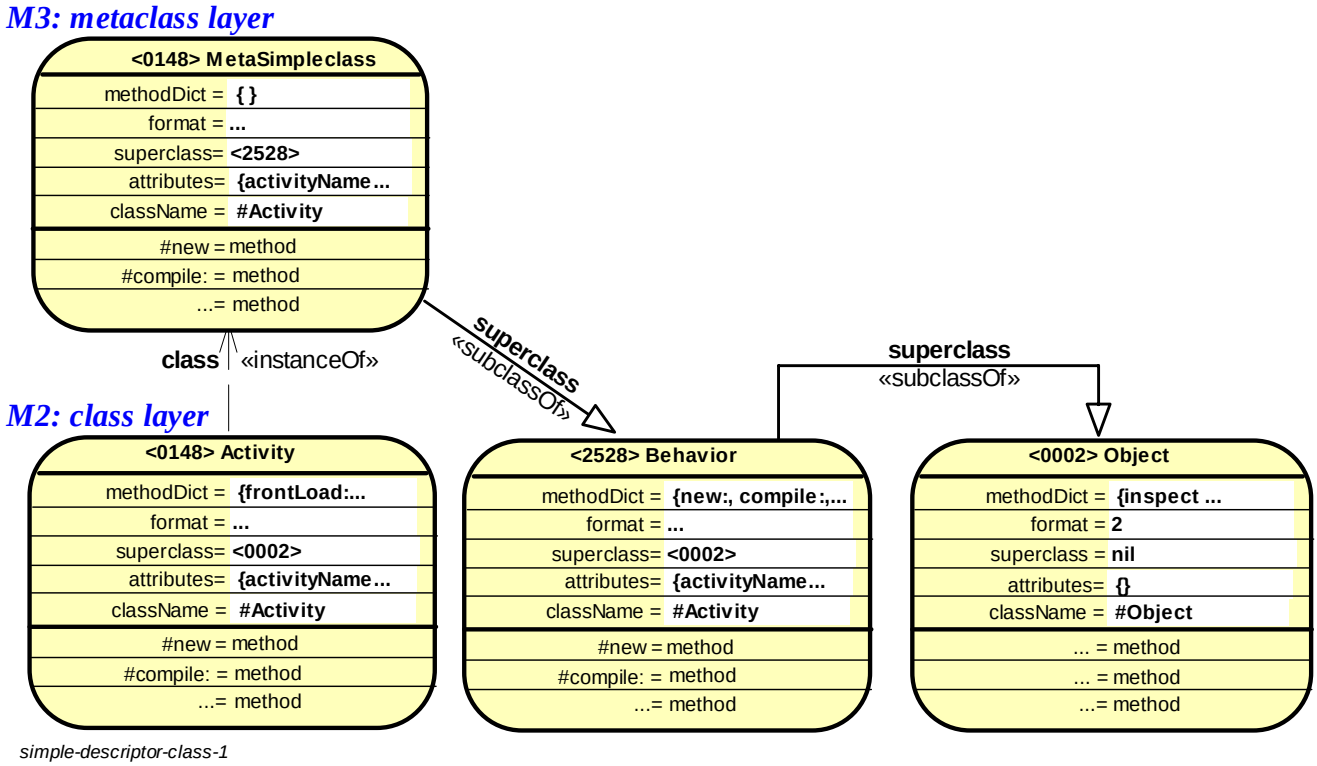
Part D: The Metaclasses

D.1: The MetaSimpleclass

i: The implementation of the Activity class object

Every object is an instance of some class. The class of a class object is called a *metaclass*. It is a rich source of confusion that the features of the <2232>actD object is given by the descriptor objects shown in figure 21 on page 28, while the features of the <0148>Activity class object is given by a different descriptor, shown in figure 29.

Figure 29: Object Descriptor for the class <148> Activity



simple-descriptor-class-1

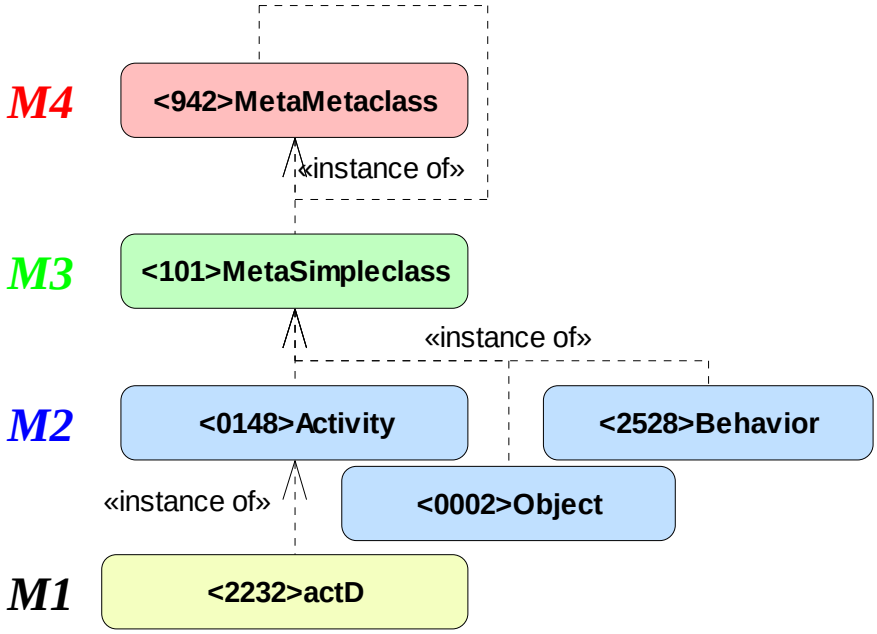
Note that the class object, <0148>Activity, responds to its own messages such as new and compile:. The methods corresponding to these messages are stored in the class of the class, <0148>Metaclass (or actually its superclass, <2528>Behavior). The metaclass is not part of the <2232>actD descriptor in figure 21 on page 28 since it in no way influences this object's features.

The toolmaker challenge is to exploit the power of flexible instantiation without confusing the programmer.

ii: The instantiation and inheritance relationships

Two additional projections of of example object are of interest. Figure 29 shows the instantiation architecture of our example object. This architecture is an essential part of the <2232>actD semantics.

Figure 30: The example instantiation architecture

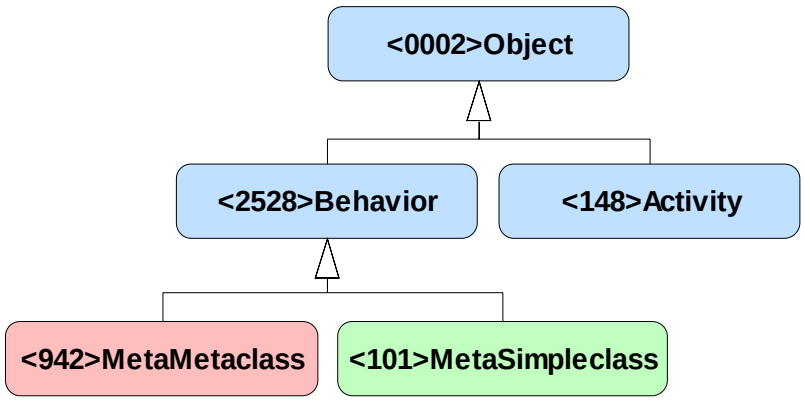


Instantiation9.sdr

Common languages such as Java have a single, buildt-in metaclass. We could have made BabyUML equally constrained by letting <101>MetaSimpleclass be an instance of itself, thus terminating the instantiation chain. We want BabyUML to be extensible with different kinds of classes and different IDEs. We therefore added an additional layer, the <942>MetaMetaclass layer. BabyUML can thus be extended by the addition of new metaclasses. Tool writers can leverage this power to create new concepts.

Class inheritance is a very powerful device for code reuse and code sharing. Figure 31 shows the inheritance structure initially chosen for the current example. We see that it bears no relationship to the instantiation structure of figure 30. The inheritance structure can be refactored without changing the system semantics. This particular solution is, therefore, relatively unimportant.

Figure 31: The example class inheritance structure.



Inheritance8

The human mind is well equipped for understanding a tree structure, but it usually finds it harder to handle two of them simultaneously. BabyUML gains its power and extensibility from its four layered instantiation architecture, but application programmers should still only see the familiar class inheritance. The toolmaker exploits the power of metaclasses to give the programmer leverage through tools such as the object browser of figure 22 on page 29.

D.2: The MetaSimpleComponent

xxxxxxxxxx

D.3: The MetaMasteredComponent

D.4: The MetaDeclarativeComponent

Part E: Windup

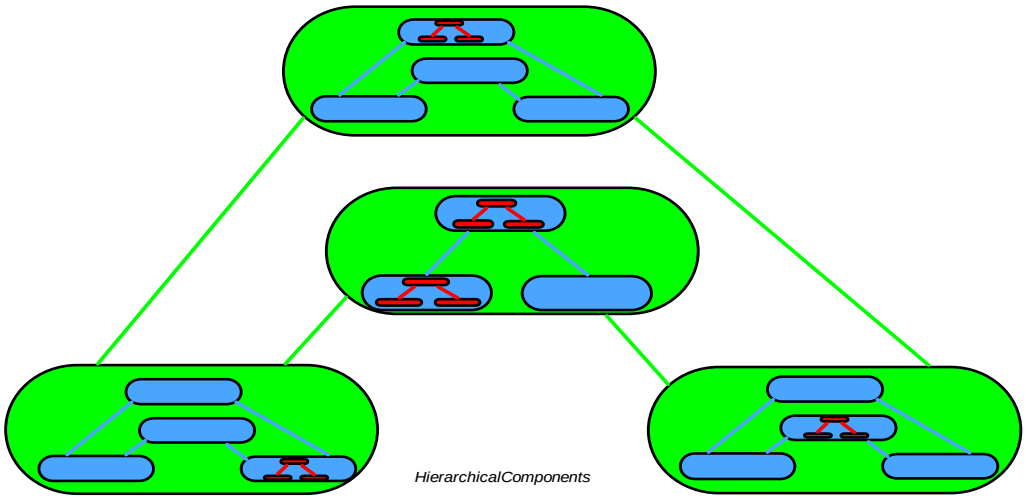
E.1: Conclusion

So, what's new? What's new is that we refuse to deal with systems that are beyond human comprehension. What's old is the recognition that we cannot test quality into a product, the only solution is to insist that our systems shall be *"so simple that there are obviously no deficiencies".* The purpose of testing is to confirm that we have done a solid job or, in rare cases, to warn us that we need to start afresh.

Our main proposal is that we shift our focus from individual objects and classes to the notion of collaborating objects. We suggest that objects be clustered into hierarchical components where each component encapsulates state and behavior, where each component is sufficiently simple to be comprehensible, and where the surface area between components is explicit and under full programmer control. (Illustrated in figure 32).

We have also adapted the UML merge relationship so that we can deal with *conceptual objects* that have identity and where the class hierarchy appear flattened into the object itself.

Figure 32: Hierarchical components for comprehensible systems.



We have exemplified our ideas with three kinds of components; other kinds can be added as needed. The *SimpleComponent* is applicable for components with few member objects and with simple interactions. The *MasteredComponent* permits more complex interaction patterns because the interaction code is lifted out of the members and made explicit in a *maestro* object. Finally, the *DeclarativeComponent* permits even more complexity because the member objects are managed by a "database" that is local to the component instance. This solution ensures referential integrity, supports persistent objects if needed, and enforces disciplined member objects.

There is a neat kind of completeness in the declarative component. *Storage-centered*: inside the component, its member objects are stored in a private micro database. *Communication-centered*: the members interact in an orderly manner to achieve a required result. *CPU-centered*: members execute methods according to their nature. The whole arrangement is recursive; a member may, on closer study, turn out to be an inner component with its own, local database, etc.

I am implementing BabyUML, a laboratory for experimenting with the new discipline of programming.

BabyUML is a *stored program object computer*; programs exist as class and other objects that exist as regular objects within components.

BabyUML supports two mechanisms for reuse. The first is that components can be wired together by *linking their ports*. The second is by *merging components*, making the contents of a merged component effectively becoming an integral part of a receiving component. Subclassing and instantiation are object relationships that are supported by the component merge.

BabyUML implements a new metaclass architecture. It is described in ???section A.3: *Introduction to BabyUML* (page 10)???. It shows how object instantiation differs from class inheritance and how BabyUML can be extended by adding new metaclasses. It also shows that each metaclass can have its own interactive development environment, making BabyUML a truly multi-language discipline. (The components discussed in this paper will all be implemented as metaclasses in the BabyUML core).

We stated a number of success criteria in the introduction. We now revisit those criteria and discuss if and how they are met by the BabyUML programming discipline.

1. *Master complexity.* Software shall be transparent so that it can be mastered by the human brain.
   - BabyUML applies a strategy of divide and conquer to ensure that the brain only has to consider a limited number of entities at the same time. Issues will be localized and a reader of the code is protected against many subtle surprises. (Clumsy code can be written in any language, but BabyUML makes it easier to write good, readable code.)
2. *Global.* The programming discipline should scale, possibly by being recursive. It should be applicable at all levels from global architecture to local detail.
   - The notion of a component is recursive and applies to all levels from the global to the innermost local.
3. *Safe and secure.* Safety and security issues are getting increasingly important. Protection support must be an essential concern throughout the programming discipline.
   - Safety and security has not been discussed here, but the component seems a natural unit for enclosure within a firewall. Safety and security must be the primary topic during the implementation of a production version of BabyUML.
4. *Controlled program evolution.* Requirements, design and code will evolve throughout the program lifetime. The discipline shall help maintain consistency between all three throughout the system life cycle.
   - BabyUML is a stored program object computer. Program and data objects are all first class citizens in the object space, and methodologies can equally support the updating of programs and data.
5. *Support software reuse and interchange.* The discipline shall support the definition of software units that can be adapted for reuse in a variety of different contexts.
   - The component is an encapsulated unit of objects with well-defined access points. The DeclarativeComponent supports persistent objects and the sharing of component information through a database.

At the time of writing, all these claims express unproven hopes. Only extended experience can substantiate them.

E.2: Further work

- UML components are characterized by their provided and required interfaces. So far, we have only considered the provided interfaces and a high priority task is to make required interfaces equally controlled and visible.
- Metaclasses and tools for the three kinds of components need to be implemented.
- Better tools for inspecting and programming. The component development tools are natural early component implementations.
- Security and safety. These important topics can only be treated properly when a first version of the component metaclasses are in place.
- Persistence, integration with database products.
- Optimization. In a private communication, Dave Thomas pointed out that object, components and other high level constructs need not exist in the running system. The running system can be optimized and the high level constructs need only exist as illusions created by higher layers of software.
- Parallel processes could be supported in mastered components with the asynchronous member interaction.

References

[Blue book] Goldberg and Robson: Smalltalk-80. The language and implementation. Addison-Wesley 1983. ISBN 0-201-11371-6

[Dijkstra] Edsger Dijkstra: *A Discipline of Programming*, 1976

[DOI] *The Digital Object Identifier System*. The International DOI Foundation http://www.doi.org OR UUID?????

[ECOOP-04] Trygve Reenskaug: *Empowering People with BabyUML: A sixth Generation Programming Language*. Opening talk, ECOOP 2004, Oslo. http://heim.ifi.uio.no/~trygver/2004/ECOOP-04/EcoopHandout.pdf

[Eng62] Douglas C. Engelbart: *Augmenting Human Intellect: A Conceptual Framework.* Summary Report AFOSR-3223 under Contract AF 49(638)-1024, SRI Project 3578 for Air Force Office of Scientific Research, Stanford Research Institute, Menlo Park, Ca., October 1962:

[GOF] Gamma et.al.: *Design Patterns.* Addison Wesley1995

[Hay-03] David Hay: *What Exactly IS a Data Model?* DM Review Magazine, February 2003 Issue

[Hy-60] T. Hysing: On the Use of Numerical Methods in the Design and Manufacture of Ships. Proceedings of the First International Congress of the International Federation of Automatic Control. Moscow, 1960. Butterworths, London.

[Norman88] Donald A. Norman: *The Design of Everyday Things.* Doubleday, 1988

[ODMG] Cattell, Barry: *The Object Data Standard: ODMG 3.0.* Academic Press, London, 2000. ISBN 1-55860-647-4

[OORAM] Trygve Reenskaug: *Working with objects. The OOram Software Engineering Method.* Manning/Prentice Hall 1996. ISBN 0-13-452930-8.
Out of print. Late draft may be downloaded here .PDF
also
Trygve Reenskaug et.al.: *OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems.* JOOP 27-41 (October 1992)

[Pawson04] Richard Pawson: *Naked Objects.* PhD thesis, Trinity College, Dublin, 2004.

[Ree-73] Trygve Reenskaug: *Administrative control in the shipyard.* ICCAS conference, Tokyo, 1973. Scanned by the author July 2003 to http://heim.ifi.uio.no/~trygver/1973/iccas/1973-08-ICCAS.pdf

[Schärli-03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz and Andrew Black, *Traits: Composable Units of Behavior*, Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming), LNCS, vol. 2743, Springer Verlag, July 2003, pp. 248—274.

[UML] Unified Modeling Language: Superstructure. Version 2.0. Object Management Group. http://www.omg.org

[Web Services] Web Services Architecture http://www.w3.org/TR/ws-arch/

[Kay77] Kay, Alan, *Microelectronics and the Personal Computer*, Scientific American, September 1977, p. 244.

[Baby] Kulwinder S. Gill: The Manchester Small Scale Experimental Machine-"The Baby" http://www.computer50.org/mark1/new.baby.htm

[Facit] Facit EDB 3 computer and ECM 64 carousell. Some documents are archived at The Norwegian Museum of Science and Technology, Autokon archive, Box #1.

[HyRee-63] Thomas Hysing, Trygve Reenskaug: A system for computer plate preparation. Numerical methods applied to shipbuilding. A NATO Advanced Study Institute organized by Central Institute for Industrial Research and Det norske Veritas. Oslo-Bergen 1963. pp.324ff
A copy of the proceedings is archived at The Norwegian Museum of Science and Technology, Autokon archive, Box #2.

About the author

Trygve Reenskaug, prof.em.,
Department of informatics,
University of Oslo, Norway.

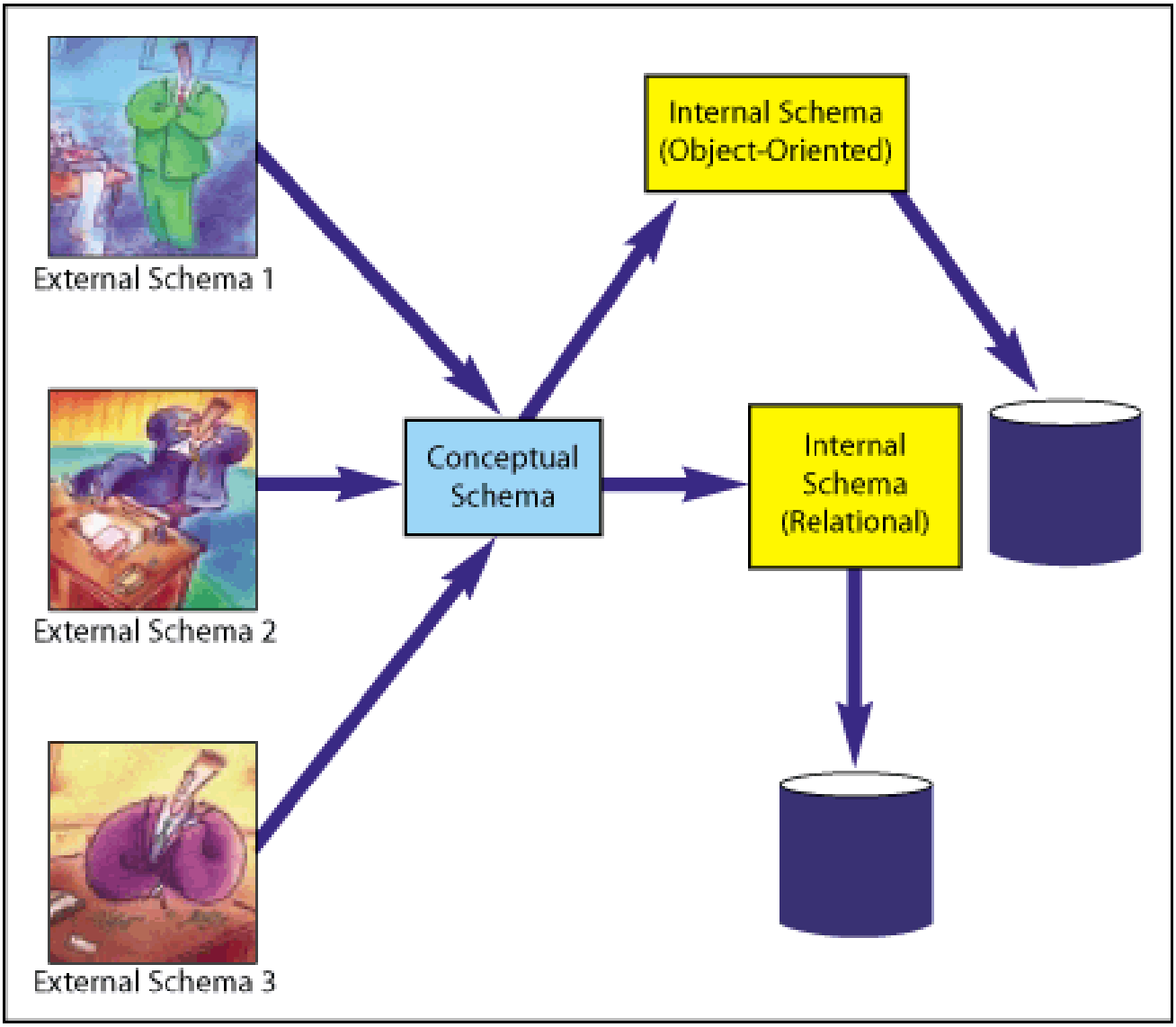
mailto: trygver <at> ifi.uio.no
http://heim.ifi.uio.no/~trygver

Appendix 1: The Declarative Component implementation

Declarative components are implemented according to the three schema architecture as illustrated in [Hay-03]. We interpret this architecture as follows:

- The conceptual schema defines the component universe of discourse with all its legal members and structures.
- The external schemas define the collaboration roles, i.e., the objects that realize the component's provided operations. Their external mappings bind them to the conceptual schema by suitable views (queries).
- The internal schema describes how the schema is implemented in the component.

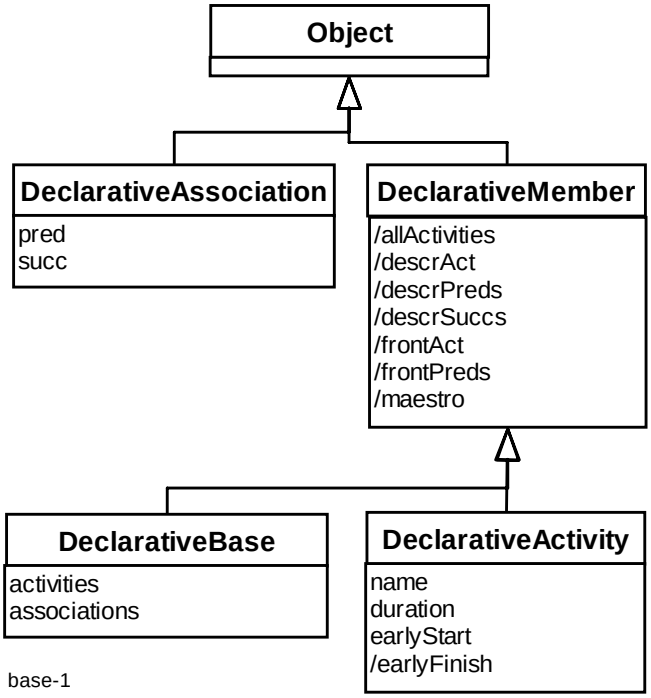
Figure 33: Three Schema Architecture (from [Hay-03])



We will now describe the sampleNet implementation as a declarative component.

The conceptual schema is given in figure 16 on page 22. Internally, the schema is implemented by two classes as shown in figure 34. We see that we have separated structure information from the activity attributes. This guarantees structural integrity and prevents smart tricks at the local activity level.

Figure 34: The internal schema.



The class definitions could be generated from the above class diagrams:

```
Object subclass: #DeclarativeAssociation
    instanceVariableNames: 'pred succ'.
Object subclass: #DeclarativeMember
    instanceVariableNames: ''.
```

The attributes of the DeclarativeMember class are all variables in the current context. They are shown as being derived, i.e., computed. (In this first experiment, the context variables are kept in a pool variable, but this will be different in future versions.)

```
DeclarativeMember subclass: #DeclarativeBase
    instanceVariableNames: 'activities associations'
```

and the “real stuff”:

```
DeclarativeMember subclass: #DeclarativeActivity
    instanceVariableNames: 'name duration earlyStart'
```

In addition, we need to define the maestro:

```
DeclarativeMember subclass: #DeclarativeMaestro
    instanceVariableNames: 'base'
```

i: activityNamed: aString duration: anInteger

The creation of a new activity is done by a data definition call on the database:

```
DeclarativeBase>>addAssociationFrom: predNam to: succNam
    (associations
        detect:
            [:assoc | assoc pred name = predNam
            and: [assoc succ name = succNam]]
        ifNone: [nil])
    ifNil:
        [associations add:
            (DeclarativeAssociation new initialize
                pred: (self activityNamed: predNam)
```

```
                succ: (self activityNamed: succNam)
        )]
```

This is called from the maestro:

```
DeclarativeMaestro>>newActivity: actNam duration: dur
    base insertIntoActivities: #(#name #duration) values: {actNam. dur.}
```

ii: addDependentFrom: actName1 to: actName2

This is also a data definition operation on the database:

```
DeclarativeBase>>addAssociationFrom: predNam to: succNam
    (associations detect: [:assoc | assoc pred name = predNam and: [assoc succ name = succNam]] ifNone: [nil])
    ifNil:
        [associations add:
            (DeclarativeAssociation new initialize
                pred: (self activityNamed: predNam)
                succ: (self activityNamed: succNam)
        )]
```

This is called from the maestro:

```
DeclarativeMaestro>>addDependencyFrom: predNam to: succNam
    base addAssociationFrom: predNam to: succNam
```

iii: frontload: firstWeek

The sequence diagram of figure 19 on page 24 defines the maestro method. The external mapping remains to map roles to component members.

```
DeclarativeBase>>inActivitiesContextDo: aBlock
    self allActivities: activities.
*   ^aBlock value.
```

and the main action:

```
DeclarativeBase>>inFrontContextRepeat: aBlock
    | frontActs |
    [frontActs := activities select:
        [:act |
        act earlyStart isNil
            and:
                [(associations select: [:assoc | assoc succ = act and: [assoc pred earlyStart isNil]
                ] ) isEmpty]
        ].
    frontActs notEmpty
    ]
whileTrue:
    [self frontAct: frontActs anyOne.
    self frontPreds: (self predecessorsOf: frontAct) asArray.
    aBlock value].
```

The current bindings of ***frontAct*** and ***frontPeds*** is now known throughout the conext and can be referenced as derived attributes in the code:

```
DeclarativeMaestro>>frontload: firstWeek
    base inActivitiesContextDo:
        [self allActivities do: [:act | act resetFrontload]
        ].
    base inFrontContextRepeat:
        [ self frontAct frontload: firstWeek.
        ].
```

Note the moving context; base inFrontContextRepeat: creates different bindings according to the current state of the activities.

The code in the activity objects is correspondingly simple:

```
DeclarativeActivity>>resetFrontload
    self earlyStart: nil.
```

and:

```
DeclarativeActivity>>frontload: firstWeek
    self earlyStart: firstWeek.
    self frontPreds do: [:pred | self earlyStart: (self earlyStart max: (pred earlyFinish + 1))].
```

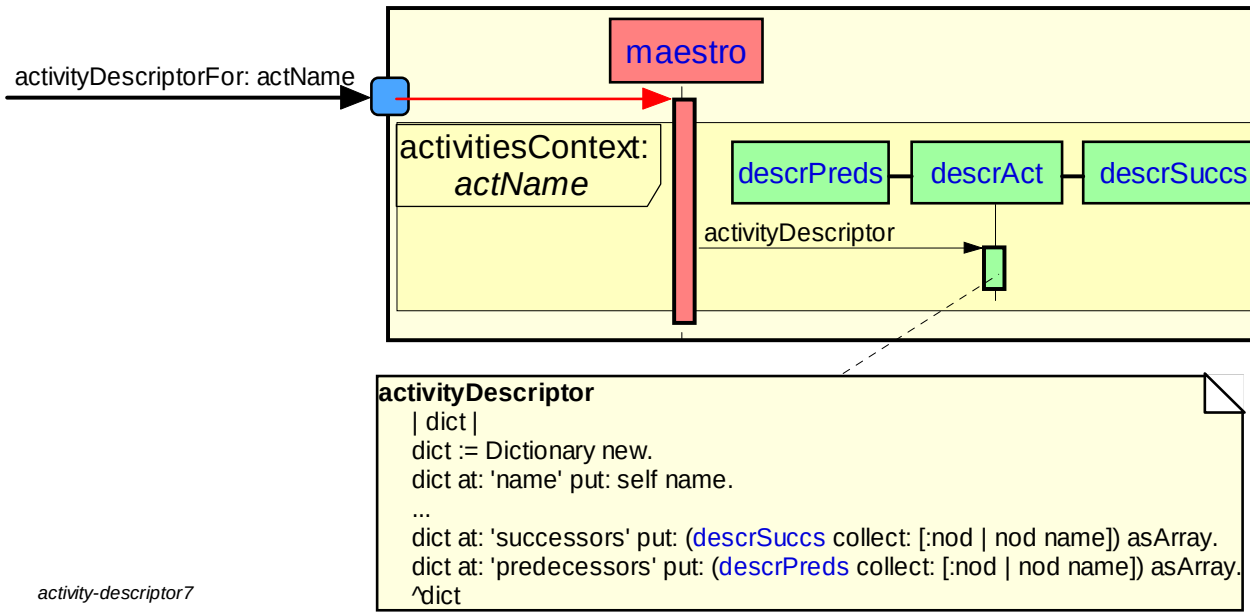
iv: activityNames

This is implemented as a simple database query:

```
DeclarativeMaestro>>activityNamesSorted
    ^base activityNamesSorted
DeclarativeBase>>activityNamesSorted
    ^(activities collect: [:act | act name]) asSortedCollection asArray
```

v: activityDescriptorFor: actName

Figure 35: Create the activity descriptor.



activity-descriptor7

```
DeclarativeMaestro>>activityDescriptorFor: actNam
    base inContextForActivityDescriptor: actNam do:
        [^ self descrAct activityDescriptor].
DeclarativeActivity>>activityDescriptor
    | dict |
    dict := Dictionary new.
    dict at: 'name' put: self name.
    dict at: 'duration' put: self duration.
    dict at: 'earlyStart' put: self earlyStart.
    dict at: 'successors' put: (self descrSuccs collect: [:nod | nod name]) asArray.
    dict at: 'predecessors' put: (self descrPreds collect: [:nod | nod name]) asArray.
    ^dict
```

Appendix 2: Component Merge

The following is an extract from [UML]:

7.3.40 PackageMerge (from Kernel)

A package merge defines how the contents of one package are extended by the contents of another package.

Description

A package merge is a directed relationship between two packages, that indicates that the contents of the two packages are to be combined. It is very similar to Generalization in the sense that the source element conceptually adds the characteristics of the target element to its own characteristics resulting in an element that combines the characteristics of both.

This mechanism should be used when elements defined in different packages have the same name and are intended to represent the same concept. Most often it is used to provide different definitions of a given concept for different purposes, starting from a common base definition. A given base concept is extended in increments, with each increment defined in a separate merged package. By selecting which increments to merge, it is possible to obtain a custom definition of a concept for a specific end. Package merge is particularly useful in meta-modeling and is extensively used in the definition of the UML metamodel.

Conceptually, a package merge can be viewed as an operation that takes the contents of two packages and produces a new package that combines the contents of the packages involved in the merge. In terms of model semantics, there is no difference between a model with explicit package merges, and a model in which all the merges have been performed.

Associations

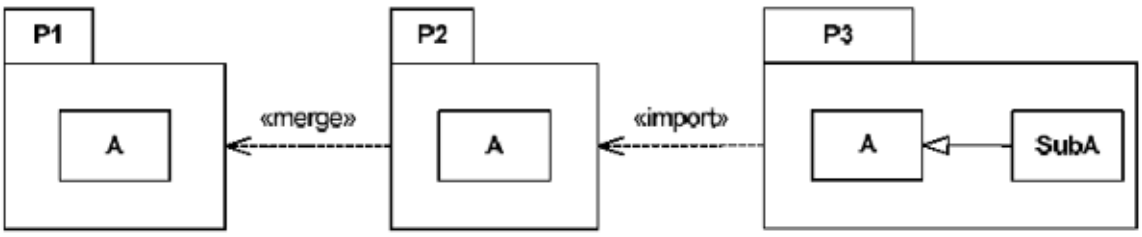
- mergedPackage: Package [1] References the Package that is to be merged with the receiving package of the Package-Merge. Subsets DirectedRelationship::target.
- receivingPackage: Package [1] References the Package that is being extended with the contents of the merged package of the PackageMerge. Subsets Element::owner and DirectedRelationship::source.

Semantics

A package merge between two packages implies a set of transformations, whereby the contents of the package to be merged are combined with the contents of the receiving package. In cases in which certain elements in the two packages represent the same entity, their contents are (conceptually) merged into a single resulting element according to the formal rules of package merge specified below.

As with Generalization, a package merge between two packages in a model merely implies these transformations, but the results are not themselves included in the model. Nevertheless, the receiving package and its contents are deemed to represent the result of the merge, in the same way that a subclass of a class represents the aggregation of features of all of its superclasses (and not merely the increment added by the class). Thus, within a model, any reference to a model element contained in the receiving package implies a reference to the results of the merge rather than to the increment that is physically contained in that package. This is illustrated by the example in figure 36 in which package P1 and package P2 both define different increments of the same class A (identified as P1::A and P2::A respectively). Package P2 merges the contents of package P1, which implies the merging of increment P1::A into increment P2::A. Package P3 imports the contents of P2 so that it can define a subclass of A called SubA. In this case, element A in package P3 (P3::A) represents the result of the merge of P1::A into P2::A and not just the increment P2::A. Note that, if another package were to import P1, then a reference to A in the importing package would represent the increment P1::A rather than the A resulting from merge.

Figure 36: Illustration of the meaning of package merge (UML figure 63)



To understand the rules of package merge, it is necessary to clearly distinguish between three distinct entities: the merged increment (e.g., P1::A in figure 36), the receiving increment (e.g., P2::A), and the result of the merge transformations. The main difficulty comes from the fact that the receiving package and its contents represents both the operand and the results of the package merge, depending on the context in which they are considered. For example, in figure 36 , with respect to the package merge operation, P2 represents the increment that is an operand for the merge. However, with respect to the import operation, P2 represents the result of the merge. This dual interpretation of the same model element can be confusing, so it is useful to introduce the following terminology that aids understanding:

- *merged package* - the first operand of the merge, that is, the package that is to be merged into the receiving package (this is the package that is the target of the merge arrow in the diagrams).
- *receiving package* - the second operand of the merge, that is, the package that, conceptually, contains the results of the merge (and which is the source of the merge arrow in the diagrams). However, this term is used to refer to the package and its contents before the merge transformations have been performed.
- *resulting package* - the package that, conceptually, contains the results of the merge. In the model, this is, of course, the same package as the receiving package, but this particular term is used to refer to the package and its contents after the merge has been performed.
- *merged element* - refers to a model element that exists in the merged package.
- *receiving element* - is a model element in the receiving package. If the element has a matching merged element, the two are combined to produce the resulting element (see below). This term is used to refer to the element before the merge has been performed (i.e., the increment itself rather than the result).
- *resulting element* - is a model element in the resulting package after the merge was performed. For receiving elements that have a matching merged element, this is the same element as the receiving element, but in the state after the merge was performed. For merged elements that have no matching receiving element, this is the merged element. For receiving elements that have no matching merged element, this is the same as the receiving element.
- *element type* - refers to the type of any kind of TypedElement, such as the type of a Parameter or StructuralFeature
- *element metatype* - is the MOF type of a model element (e.g., Classifier, Association, Feature).

This terminology is based on a conceptual view of package merge that is represented by the schematic diagram in figure 37 (NB: this is not a UML diagram). The owned elements of packages A and B are all incorporated into the namespace of package B. However, it is important to emphasize that this view is merely a convenience for describing the semantics of package merge and is not reflected in the repository model, that is, the physical model itself is not transformed in any way by the presence of package merges.

Figure 37: Conceptual view of the package merge semantics (UML figure 64)