A Rudimentary UML Virtual Machine as a Smalltalk Extension

Trygve Reenskaug

Draft of February 24, 2002 (book4)

***Abstract*.**

The UML metamodel defines the abstract syntax of well-formed model. A UML model is a structure of interrelated objects that conform to this syntax.

A Smalltalk world is a structure of interrelated objects that reside in a Smalltalk Virtual Machine (VM). Anything and everything of interest exists as objects in this VM. Things in the application domain exist as objects. Classes and metaclasses exist as objects. Parsers, compilers, inspectors, and debuggers exist as ensembles of interrelated objects just as any other applications.

There are three kinds of structures of interest in a Smalltalk VM.

- Collaboration structure. *How objects are interconnected and how they interoperate to realize a certain use case.*
- Subclass/superclass structure. *How classes inherit from other classes. (Standard Smalltalk has single inheritance)*
- Instantiation structure. *Every object is an instance of a class. So every class object is an instance of some class, which is an instance of some class, etc.*

I have implemented the *Mike 11 years old* *example in Smalltalk and inspected some interesting objects. I find application objects, class objects, metaclass objects and possibly also a metametaclass object.*

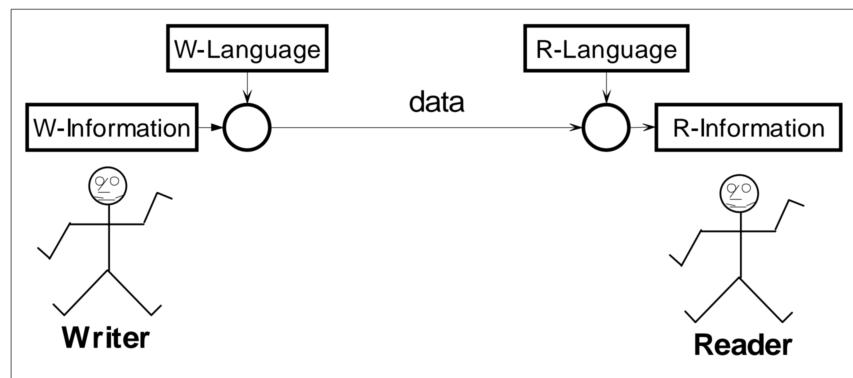
*The analogy to UML is striking. I have found I cannot see through the details of a UML Virtual Machine (*UML-VM*) without actually writing one. The result has been enlightening. This report describes my first, rudimentary* UML-VM*, the steps that led up to it and the conclusions I could draw from it.*

This report has been written with Adobe FrameMaker®. Illustrations have been drawn with SmartDraw® and transferred to the document in WMF format. The document was finally transferred to PDF format with Adobe Acrobat®

1. Introduction and motivation

The definition of the UML semantics in [U2P] assumes that the reader is familiar with UML, OCL, and American English. But as in all human communication, the communication cannot be perfectly precise. Whenever in doubt, a reader will draw on his background knowledge and preconceived notions to interpret the words and diagrams. The purpose of this report is to make some of my own assumptions explicit and see how they influence my interpretation of the UML semantics document. The result is a clarification and exemplification of the UML Language Architecture which I hope is consistent with the notions of the [U2P]team.

Figure 1: Inter-human communication is inherently distorted.



Whenever a writer writes something, he uses his knowledge of the language to codify his ideas. Whenever a reader reads something, he uses his knowledge of the language to decode the incoming data and reconstruct a meaning in his brain. It would be strange indeed if the reader's language were in all respects identical to the writer's language, so we would expect the reader's understanding of the subject matter to be somewhat different from the writer's intentions.. This is illustrated in figure 1.

I feel this problem strongly when I try to understand the [U2P]UML layered architecture in [U2P]chapter 1. The chapter is still in its early stages and opens many questions in my mind. I'll start with figure 1-3. It is repeated for your convenience in figure 3 below. My first uncertainty is about what I find on level M0. Is it the boy called Mike? Is it an object in a computer's memory that describes the boy? Or is it a model of the object that describes Mike?

One argument is that the real world is the target for all our modeling, it should be shown somewhere in the architecture. An opposite argument is that we are describing five layers of *modeling;* there is no room for real world things such as people and memory bits in this architecture. Does it matter, or should we be satisfied with a tabloid description of the fundamental UML architecture? I believe it does matter. The architecture is the foundation of the whole UML edifice. We need a shared understanding of this architecture to realize a shared understanding.

The real meaning of the UML semantics is to define the set of all possible consistent, coherent, and well formed structures of UML model objects. Some objects model user objects, other objects model classes, namespaces, attributes, operations, packets, activities, etc. The nature of these objects is the subject of this report.

Our goal is not only to document these objects, but to implement a rudimentary UML-VM and observe the relevant application and model objects. From this we hope to achieve a deeper understanding of the nature of UML models.

2. Summary and Conclusion

My basic assumption is that the Unified Modeling Language is a pure object oriented language. This means that the UML definition document defines all permissible well-formed structures of UML model objects. It is, therefore, of interest to study the nature of these physical objects in addition to their abstract specification.

Most object oriented languages such as Java and UML are class based. This means that classes are defined to hold the properties that are common to all the instances of the class. Objects exist at run time on a substrate of a *Virtual Machine (VM)*. The Java VM is one example, the Smalltalk VM is another. In both, it is possible to interrupt the execution and inspect the objects as they exist at the time of interruption.

In addition to the application objects themselves, a VM must also contain information about classes and metaclasses. Classes are needed to hold the information needed for instantiation and the methods shared by all instances and other information. Metaclasses are needed to hold corresponding information for the class objects themselves such as static methods.

Class and metaclass information is implied and hidden in most VMs. Smalltalk is here an exception. In Smalltalk, classes, metaclasses, methods, and all other information of interest are explicitly available as visible objects in the Smalltalk VM (ST-VM). The mechanisms for class instantiation are likewise explicit and visible and can be modified by the programmer. This makes the ST-VM ideally suited for experiments with a UML Virtual Machine (UML-VM). Smalltalk has, therefore, been chosen as the foundation for this project. It should be noted that this choice in no way deters from the general applicability of the project results. Smalltalk merely makes these results more concrete and visible.

The UML layered architecture could be taken seriously, or it could be viewed it as an "artist's impression" with no deep significance. The whimsical view is strongly supported in the body of the definitions[U2P] because there is no explicit definition of the «instanceOf» hierarchy. I have chosen to take the serious view. This has been strengthened by the experiments because the «instanceOf» relationships give significant insights into the nature of the object systems. UML models consisting of a number of interconnected and interacting model objects is a case in point.

I have done two experiments in this project. In the first experiment, I programmed a simple example creating an object representing an 11 years old boy called Mike. I examined this object with its class, and followed the «instanceOf» pointers to see the instantiation hierarchy. I also followed the superclass relationships to see the inheritance hierarchy.

A most important discovery in this experiment was the discovery of the difference between the object «instanceOf» hierarchy and the subclass/superclass hierarchy. I discovered that the «instanceOf» hierarchy is an essential part of the system semantics, while the arrangement of the classes in the class inheritance hierarchy did not influence the system semantics and is more in the nature of a comment. It was a sobering thought that the elaborate organization of the UML definition with packages, classes, and inheritance is almost irrelevant for the semantics of the UML model objects and the UML-VM. (But not irrelevant for our ability to understand these specifications!)

In the second experiment, I implemented a rudimentary UML-VM. This implementation clearly and explicitly reflects the four-layer metamodel architecture. I found the implementation to be useful and illustrative for what UML is all about. It greatly helped my understanding because it gave a concrete and physical realization of these specifications.

I first implemented a root object, U::Metaclass, that represents the meta-meta-metaclass. As for all classes, the (MOF) M3 class was created by sending the message new() to the U::Metaclass object:

1. U::Metaclass.new() yields the MOF class, it is called M3::Class.
2. M3::Class.new() yields the UML class object: M2::Class.
3. M2::Class.new() yields a UML model object. Suitable messages to this object gave it its name, M1::Person, the instance variable names of its instances, ('name', 'age'), and its methods for getting and setting values for the instance variables. This was where the "classtance confusion" became very visible: The class object holds the properties of its instances, while the metaclass object holds the properties of the class object, etc. all the way up the «instanceOf» hierarchy.
4. M1::Person.new() yields the application object that after proper initialization represented Mike, 11 years old.

I now discuss the nature of the M0::Mike object. It is clearly not the boy himself; a UML class cannot possibly produce a real boy. But neither could I make it the real application object from the first experiment. In my ST-VM, all objects are real and visible. I see that the application object, ST::Mike, is an «instanceOf» the Smalltalk class ST::Person. MO::Mike is an «instanceOf» M1::Person, which is distinct from the corresponding application class. So the two worlds are related, but not identical.

One of the first activities in the first experiment was to develop a UML class diagram and a UML Message Sequence Chart for the first experiment. At the end of the experiment, I assigned a layered architecture to the «instanceOf» hierarchy. I found a layered architecture similar to the UML layered architecture, and called the layers ST0 through ST4. I made this very interesting discovery: *The UML class diagram perfectly modeled the Smalltalk classes in the ST1 layer. The MSC perfectly models the interactions that take place in the ST0 layer.*

I postulate that the fundamental difference between class modeling and role modeling is that the first describes model objects with their relationships in the M1 layer, while the latter describes model objects in the M0 layer with their static and dynamic relationships.

I have earlier had long discussions about the nature of the ClassifierRole; trying to determine if it is a class or an instance. The above postulate makes this discussion meaningless. It is not a question of what it *is*, i.e., its place in the class inheritance hierarchy, but it is a question of its position in the «instanceOf» hierarchy.

In [U2P], Class and Part are both defined as subclasses of Classifier. This makes perfect sense. But what I postulate is that they belong in different layers in the layered architecture; Class should be in the M1 layer while Part should belong in the M0 layer.

My conclusion is that the four-layer architecture of [U2P]figure 1-3 makes perfect sense. The «instanceOf» hierarchy should be defined explicitly in the body of UML 2.0. Further, the Common Core architecture of [U2P]Figures 1-1 and 1-2 also make sense, they reflect the organization in the class inheritance space.

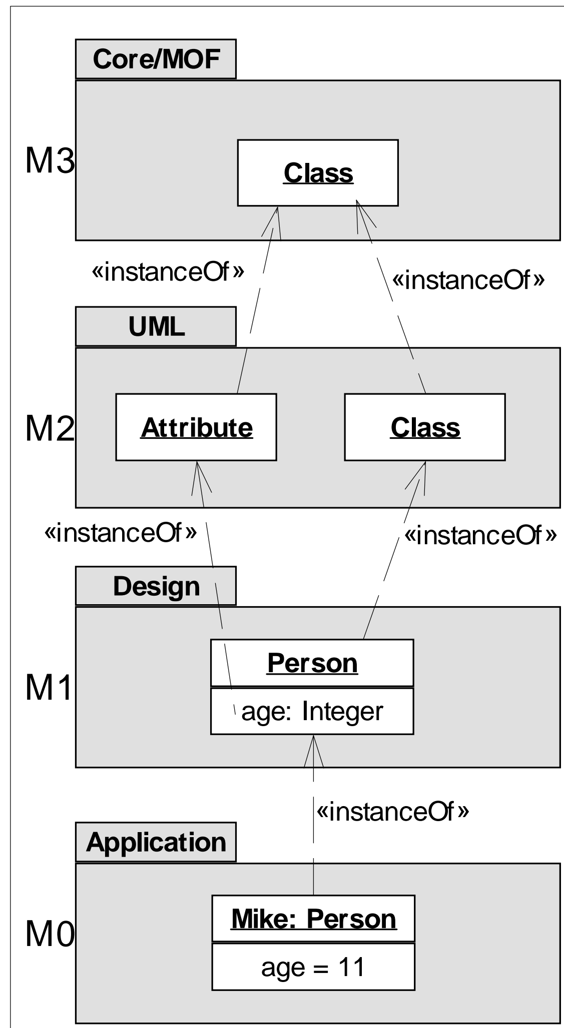
In figure 2, I propose a slight modification to the original figure 1-3.

The key is that all objects are created by some object, just as in the case of Smalltalk. The architectural layering is a layering of objects along the «instanceOf» relationships as shown in the original figure 1-3.

I have changed the name of the *Analysis layer* to *Design layer.* (There is no verb "to architect" in Webster. I have been told that an architect *designs* an architecture).

Finally, I have changes the name of the bottom layer to "Application" because the whole figure shows an example. Also, there are users of the different layers. (End users, application designers, tool builders, ...)

Figure 2: Figure 1-3 from [U2P] revised



3. First experiment: A Smalltalk implementation of the example

The purpose of this experiment is to examine in detail how objects, classes and metaclasses are handled in Smalltalk. It transpired that most of these details are artifacts of the Smalltalk library so that they can be changed by the Smalltalk programmer. Finding the details of how this is done in regular Smalltalk provides an entry point into the creation of a UML-VM within the scope of the ST-VM.

The first step in this first experiment was to implement class Person in regular Smalltalk and create the Mike instance. The experiment was organized in the following steps:

5. *Model the experiment in UML, code it in regular Smalltalk, and run the test.*
6. *Inspect and study the interesting objects resulting from this experiment.*

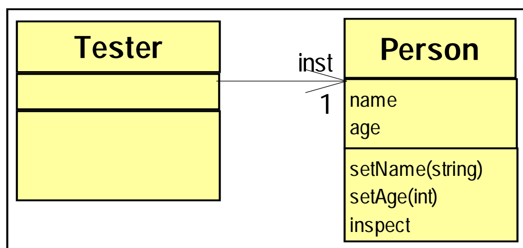
There are three kinds of structures of interest in any object system. I study them in some detail in this experiment:

7. *Study the collaboration structure*. Study how objects are interconnected and how they interoperate to realize a certain system operation (functionality).
8. *Study the specialization/generalization hierarchy*. Study how classes inherit properties from other classes. (Regular Smalltalk has single inheritance)
9. *Study the «instanceOf» hierarchy*. Every object is an instance of a class. Every class exists as an object in the ST-VM. So every class object is an instance of a metaclass, which is an instance of a metametaclass, etc.

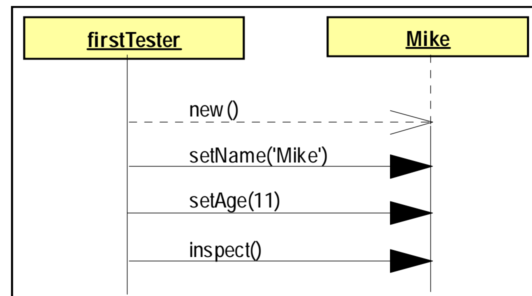
3.1. Model the experiment in UML, code it in regular Smalltalk, and run the test

Figure 3: Example UML model and Smalltalk code..

(a) Class diagram



(b) Message Sequence Chart (MSC)



(c) Smalltalk code in an informal syntax

```
class Tester extends Object {
        inst;
        runTest() {
                inst = Person.new();
                inst.setName('Mike');
                inst.setAge(11);
                inst.inspect(); }
}
class Person extends Object {
        name;
        age;
        setName(string) {
                name = string; }
        setAge(int) {
                age := int; }
}
```

The MSC shows that the new Person object, Mike, is created by sending the *new()*-message. Note that this message goes to the class object, Person, not to Mike which doesn't exist yet.

The code in figure 3(c) is written in a Java-like syntax that I have concocted in the hope that it will prove more palatable than the somewhat unconventional Smalltalk syntax.

A small point is that the inspect()-operation is defined for class Person in figure 3(a), while it is not implemented in the code in figure 3(c). inspect() is actually inherited from class Object in the code. It is not shown as a derived operation (/inspect) in the class diagram because the model does not specify this inheritance.

3.2. Inspect and study the interesting objects resulting from this experiment

The experiment is controlled by the runTest method in the firstTester object as shown in the MSC of figure 3(b):

1. firstTester sends the message new() to the class object Person. The method returns the ID of the new object so that firstTester can now send messages to it.
2. firstTester sends the setName('Mike') message to the Mike object. I now have to be very careful. The Mike object receives the message and the method is *executed* in the context of this object. But the method is *stored* in the class object Person and is retrieved from it when needed.

I open an Inspector that gives access to all interesting objects including class and metaclass objects. The three most important objects are shown in figure 4. Figure 5 on page 9 shows the complete «instanceOf» chain of objects.

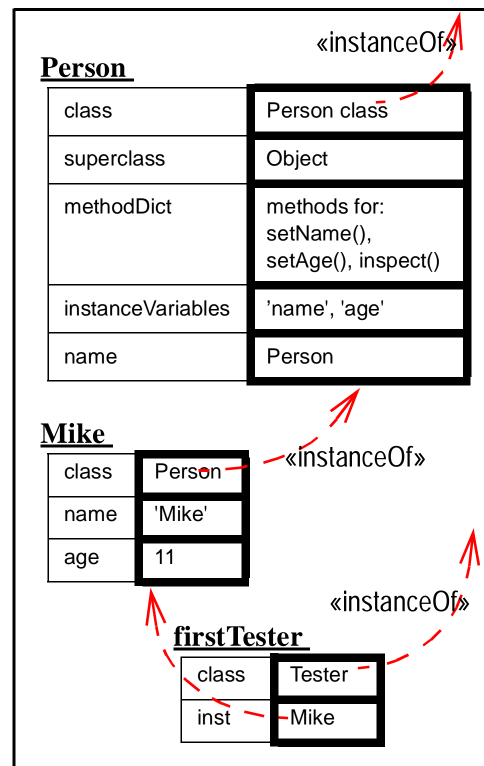


Figure 4: Three Smalltalk objects that belong to the example application.

Sources such as [BLUEBOOK] and the inspected objects illustrate some very important and powerful features of pure object systems such as Smalltalk.

1. The Smalltalk programmer sees a *uniform* world of objects that reside in the ST-VM. Anything and everything of interest exists as an object. The Mike object and the firstTester objects are obviously objects. Even the class Person exists as an object as illustrated in figure 5. Parsers, Compilers, Inspectors, Compilers, debuggers, source code, compiled methods (byte codes), all exist as objects. There is nothing else, even integers are known by their object IDs.
2. An object has *identity*. An object's identity is distinct from its attributes. The ID is unique and immutable and can be stored and communicated as a value.
3. An object exists in the computer memory as a block of values. The block header contains some information used by the ST-VM for e.g., garbage collection. The header also contains the ID of the object's class. The body of the block is an array of object IDs. This gives rise to the saying that "in Smalltalk, everything you look for is somewhere else". The objects are shown with heavy outline in figure 4. The object IDs have been replaced by their values to improve readability, and the names of the attributes are shown.
4. Objects are *encapsulated*. In Smalltalk, this means that an object can only interact with another object by sending messages to it. The internal construction of an object is invisible to other objects.
5. Objects have *state*. The state of an object is given by the combined values of the object IDs stored in the object's block of memory.

6. Objects exhibit *polymorphism*. This means that logically, every object deals with incoming messages according to the rules laid down in its own methods.
7. Every object is an instance of a *class*.

An important responsibility for the class object is to hold all information that is common to all its instances. This is an obvious efficiency device; it would be ridiculous to let every object's memory block contain a copy of such information as attribute names and method byte codes. Logically, this information is distributed among the objects as we have seen above. Technically, common information is kept outside the object. (E.g., in a class object).

The class object is a regular object with all the properties described above. In addition, it has some properties of its own:

8. The class object obeys a factory message such as basicNew(). The corresponding method calls a method such as primitive 70. It causes the ST-VM create a new object with a unique ID and a block of memory.
9. The class object responds to e.g., setName('Person'), a message that is used to give the class a name.
10. The class object responds to e.g., setInstanceVariablenames('name', 'person') that is used to set the number of attributes and their names.
11. The class object responds to getCompiler(). This permits the methods of a class to be written in the class's very own programming language.
12. It responds to setMethod(operation, method). This is used to associate operations with appropriate methods.

Figure 5: The instantiation chain of the example application.

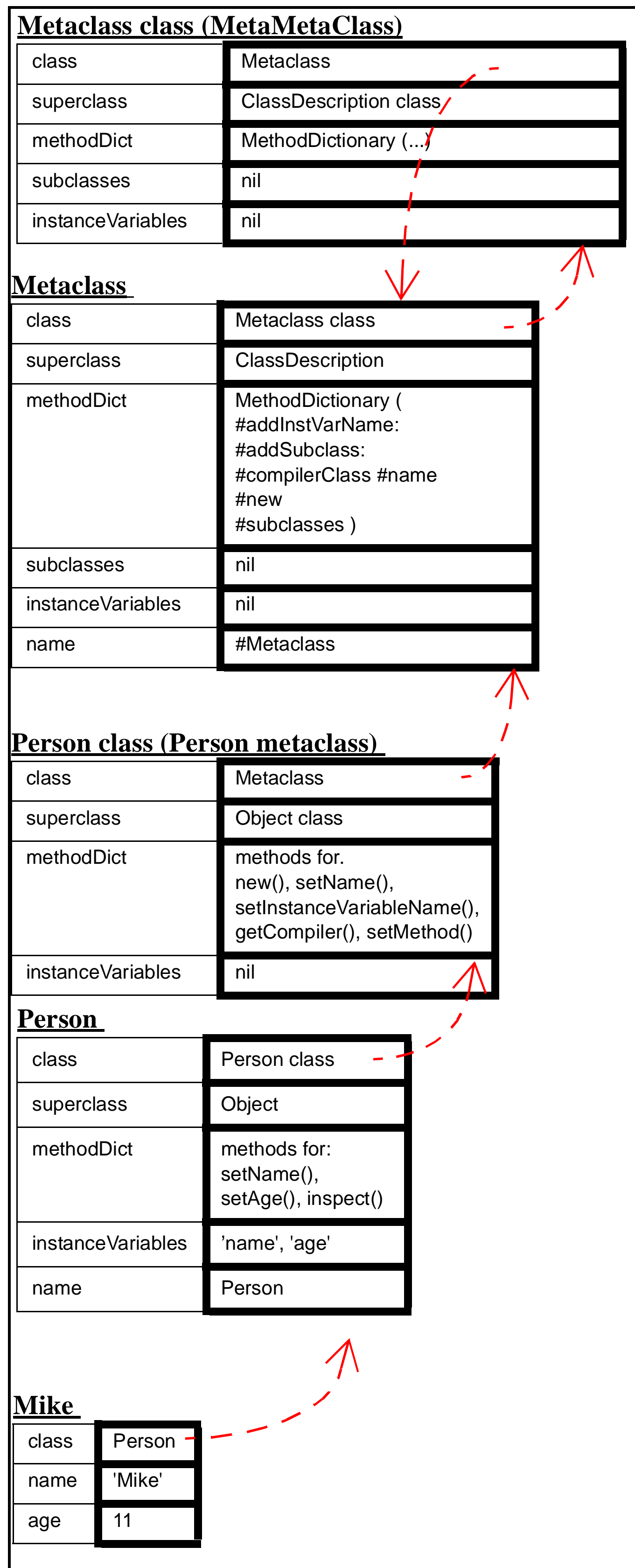


Figure 5 shows all the objects in the«instanceOf» hierarchy.

We see that the instance variables of the class object, Person, are different from those of the application object, Mike, itself. The class has a method dictionary that binds Mike's external operations to the methods. Additionally, there are the names of Mike's instance variables ('name' and 'age'), and the name of the class itself.

Person is an object, so it must be the instance of some other class. By Smalltalk convention, it is called Person class.

The Person class object is an instance of Metaclass, which is an instance of Metaclass class. The plot thickens; the potential infinite recursion is broken by the circularity that Metaclass class is an instance of Metaclass. So the two are instance of each other!

A class object holds the methods for its instances, not for the class object itself. The methods for the class object are held in the class of the class, i.e. its metaclass. And the methods for the metaclass is held by its class again, i.e., the metametaclass. This indirection can often cause confusion of the kind that is caused by indirect addressing in assembly programming. The *"classtance confusion"* is the name I often use for the class/instance confusion.

3.3. Study the collaboration structure

The object collaboration structure is a connected system of the objects that are needed to realize some function (or system operation or use case).

Kent Beck and Ward Cunningham are quoted as saying: "... no object is an island". Individual objects are only interesting to the extent they help us understand a system of interacting objects. Viewed in this light, the MSC figure 3(b) should be more interesting than the classes of figure 3(a).

One might claim that the methods are in the class, not in the instance. But this is a fallacy, the methods are always executed in the context of the instance. The fact that they are stored in the class is an obfuscating optimization.

In UML, a system of interlinked objects is modeled by a *Collaboration*. Examples of interactions are modeled by interaction diagrams such as the MSC in figure 3(b). The nature of the interaction can be modeled more precisely with various kinds of state and activity models.

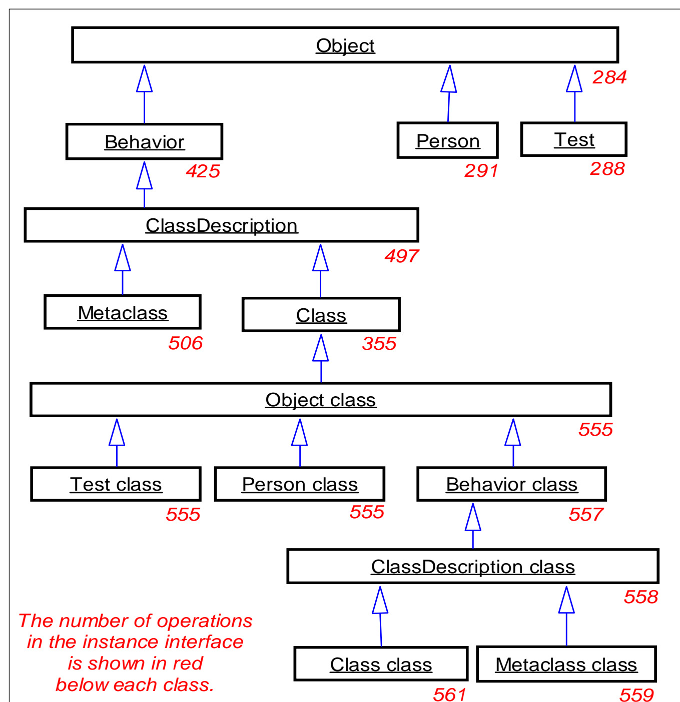
3.4. Study the specialization/generalization hierarchy

The class objects of the simplified figure 4 and extended figure 5 all have an attribute called superclass that identifies a superclass object. The class diagram of figure 6 has been created by following the superclass reference chain, starting from the objects of figure 5.

We see that in Smalltalk, class Object does not have a superclass. (The value of the superclass attribute in the object called *nil*). Smalltalk permits any number of class objects with superclass=nil, so there can be any number of disjoint class hierarchies. I believe the [U2P] class hierarchy has a single root, namely Element. (For further study and trial implementation).

It is worth noting that the arrangement of the superclass/subclass structure is irrelevant to the executing objects in the ST-VM. A class can always be replaced by a more comprehensive class where all the superclass information has been merged into the class. Class inheritance is a great code sharing device. It can also help organize the concepts as perceived by the human mind. But the arrangement of the class inheritance hierarchy does not influence the system semantics.

Figure 6: The superclass/subclass structure of the objects of figure 6.



It is a sobering thought that the elaborate organization of the UML semantics definition could be flattened so that every class description were complete and self-sufficient. The result would be an enormously more voluminous document without structure. The disadvantages are obvious. But there would also be an advantage: The complete definition of a concept would be concentrated in one place, saving the reader to search through the whole document to assemble all the partial definitions that are distributed all over the place.

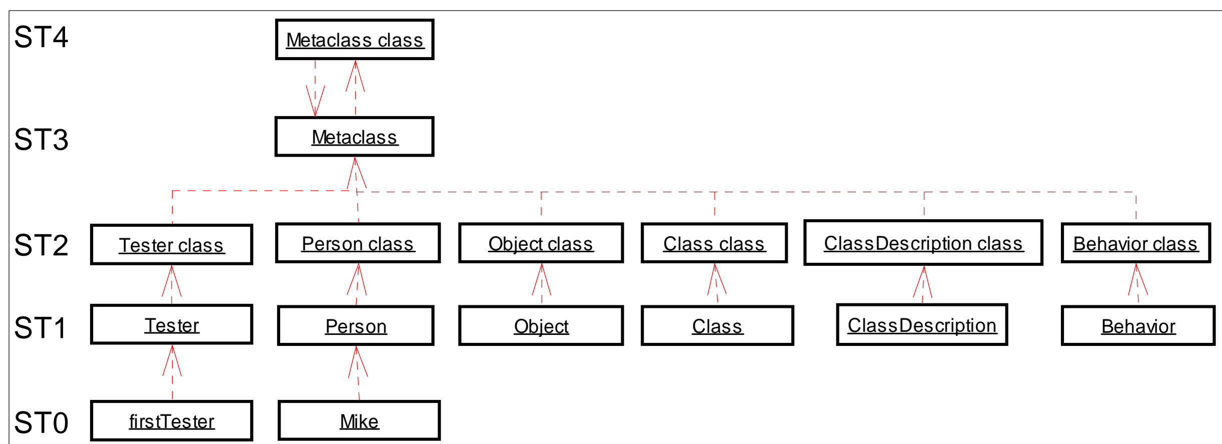
The class hierarchy shown in figure 6 is very deep, and the operation interfaces of the leaf classes is quite unmanageable. For example, instances of Class class and Metaclass class respond to 561 and 559 operations respectively. I cannot relate to this plethora of operations.

In section 4 on page 13, I describe a special version of class Metaclass that is the root of the UML-VM classes. The only way I could do that was to first understand the class creation activity in regular Smalltalk, and then modify this activity to cater for class creation in the UML-VM.

3.5. Study the «instanceOf» hierarchy

We saw in the previous section that the arrangement of the class inheritance hierarchy has no semantic significance in the Smalltalk execution model. In contrast, the «instanceOf» relationship is an essential part of the execution model. This relationship is established by the ST-VM when it creates the object. An object cannot exist without its class object since the class determines the object's behavior.

Figure 7: The «instanceOf» structure of the objects of our example.



I found the «instanceOf» structure by following the class pointer from object to object. The first experiment gave me the structure of figure 7. This structure reflects the regular Smalltalk instantiation and execution architecture.

I have defined five layers in the Smalltalk instantiation architecture as an analogy to the UML four-layer architecture:

ST0, Application objects. The objects in this layer are the Smalltalk application objects; here firstTester and Mike.

ST1, Application classes. The objects in this layer create the ST-0 objects and hold their common features; here the classes Tester, Person, Object, Class, etc.

ST2, Metaclasses. The objects in this layer create the ST-1 class objects and hold their common features; here the metaclasses Tester class, Person class, Object class, etc.

ST3, Metametaclass. The objects in this layer create the ST-2 objects and hold their common features. There is only one class in this layer, Metaclass.

ST4, Metametametaclass. The object in this layer creates the ST-3 object and holds its features. There is only one class in this layer, Metaclass class. This object is an instance of Metaclass and thus terminates the «instanceOf» structure.

It should be noted that this architecture applies to most object oriented programming languages such as Simula and Java. The main difference is that the architecture is implied and hidden in most languages while it is explicit and visible in Smalltalk. In Smalltalk, it is even possible to modify the architecture to cater for special needs such as the creation of a UML Virtual Machine that will be discussed in Section 4 on page 13. This is the reason for choosing Smalltalk as an environment suitable for embedding the UML-VM.

As an illustration, we will dream up a layered architecture for Java:

J0, Application objects. The objects in this layer are the Java application objects.

J1, Application classes. The objects in this layer are the classes that we write in the Java programming language. The Java classes exist as .class files that are interpreted by the Java run time system. We imagine that they exist in the run time system as (invisible) class objects.

J2, Metaclasses. The (invisible) objects in this layer hold Java static variables and methods, including the factory methods.

J3, Metametaclass. Who creates and manages the J2 objects?

3.6. Conclusion of the ST-VM experiment

It is interesting to see what real things are modeled by the UML models and Smalltalk code of figure 3.

The code for class Person in figure 3(c) completely specifies the class object Person. The code specifies the number of variables and their names, as does the instanceVariables variable in the Person object. The code also specifies the operations and methods, i.e., the contents of the class object's methodDict. At run time, the information from the code is found explicitly in the class object in the ST1 layer.

The rectangles in the UML class diagram of figure 3(a) similarly model the class objects Person and Tester. We see that the association from Tester to Person models the instance variable inst. The attributes of class Person are likewise models of the instVarNames variable in the Person object. This is interesting; UML Attributes and AssociationEnds seem to models of the same thing. May be it should be considered if they are the same or if they at least should have some common root. Note that all information from the class diagram is found in the ST1 layer of ST-VM.

The rectangles in the UML Message Sequence Chart of figure 3(b) clearly model the application objects since messages are sent from application object to application object, not from class to class. So the MSC models phenomena that belong in the ST0 layer. (Interaction between class objects is a different story).

To me, this is an important discovery: The code and the class diagram describe entities in the ST-1 layer. The MSC models entities and behavior model phenomena in the ST-0 layer.

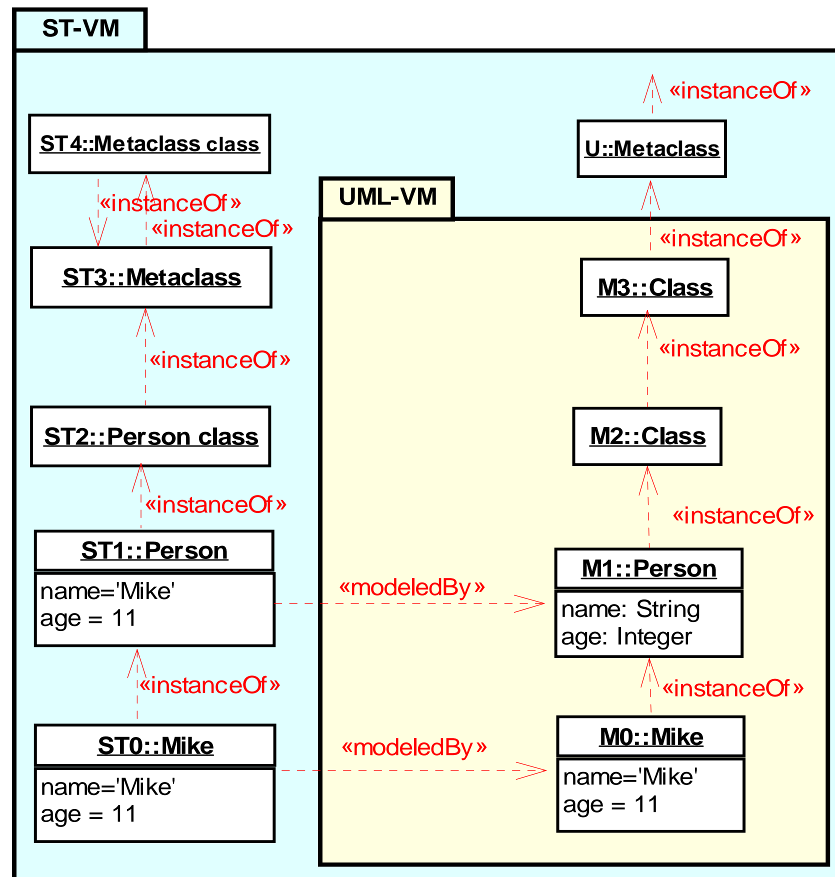
I have for many years tried to find a good explanation between build time and run time modeling. In [REE01], I suggest that there are two important perspectives on systems modeling. I called them the Class Perspective (CP) and the Role Perspective (RP). All attempts at explaining the difference between these two perspectives in terms of model classes have failed. We may now be on the track of the real difference: Models in the Class Perspective model phenomena on the ST1 layer, e.g. the class objects. Models in the Role Perspective model phenomena in the ST0 layer, e.g. the application objects. This sounds promising and is for further study.

4. Second experiment: A Rudimentary UML Virtual Machine

In the second experiment, I implemented a rudimentary UML-VM in order to study the nature of its instantiation and class inheritance hierarchies.

In figure 8, I have redrawn and simplified figure 3 on page 3 to become the design of the instantiation hierarchy of my first UML Virtual Machine (UML-VM). The regular Smalltalk implementation from the first example is drawn to the left of the UML model objects so that we can see the model objects together with the corresponding application objects.

Figure 8: Instantiation hierarchy of the first UML-VM.



In a preliminary study, I implemented the UML model objects as instances of regular Smalltalk classes. This solution worked after a fashion, but was very unsatisfactory. The worst part was that the UML model class objects were not real class objects because they lacked the primitive ability to create a new object. Instead, the UML "class" object had to enlist the help of a regular Smalltalk class with its metaclass etc. as described in the previous section.

It seemed necessary to implement a pure solution where the UML model objects were real class and metaclass objects that were capable of instantiation.

The first experiment gave a background understanding of the regular Smalltalk instantiation hierarchy that made it easy to see what must be different for the corresponding UML hierarchy. In regular Smalltalk, there is a one-to-one correspondence between class and metaclass. The class object, e.g., Person, holds the properties of the instance, e.g. Mike. The metaclass object, e.g., Person class, holds properties such as static methods and variables for the class object itself.

In the UML-VM, there are no class-specific static variables and methods. Further, all metaclasses can have many instances.

The following modifications were done to the regular Smalltalk instantiation architecture:

1. I wrote a new version of class Metaclass in order to get around the one-to-one relationship between class and metaclass. I called it class U::Metaclass.
2. The «instanceOf» hierarchy must be closed at the top. In regular Smalltalk, this is done by the classes Metaclass and Metaclass class being instances of each other. Conceptually, they must be created by

magic before we can create any other class. Here, I let the magic be outside the UML-VM by rooting the UML «instanceOf» hierarchy in a regular Smalltalk class, class U::Metaclass.

3. Smalltalk programs are commonly written in the Smalltalk programming browser. My unorthodox architecture forced me to create my UML-classes programmatically. The classes are created by methods in the Tester. (I may later create a modified browser that will support the development of UML classes within the UML-VM).

The model objects of the UML-VM are shown in figure 9.These objects are indeed a true implementation of the UML layered architecture.

Figure 9: The UML model objects of the rudimentary UML-VM as revealed by the Inspector.

M3::Class

class	U::Metaclass
superclass	U::Metaclass
methodDict	()
instanceVariables	nil
name	M3::Class

M2::Class

class	M3::Class
superclass	U::Metaclass
methodDict	()
instanceVariables	nil
name	M2::Class

M1::Person

class	M2::Class
superclass	U::Element
methodDict	(#age: #name: #name #age)
instanceVariables	('name' 'age')
name	M1::Person

M0::Mike

class	M1::Person
name	'Mike'
age	11

We see that M3::Class is an instance of U::Metaclass and also a subclass of the same class.

We also see that M2::Class is an instance of M3::Class as required by the layered architecture. There are no additional features in this rudimentary implementation, so I made M2::Class a subclass of U::Metaclass.

We now get to the UML modeling layer, M1. We see that M1::Person is an instance of M2::Class, as it should be. This is a regular class and its instances are not classes. I made M1::Person subclass of U::Element, which happens to be an empty subclass of Object. (I could not make it subclass of U::Metaclass because it would then become yet another metaclass with classes as its instances).

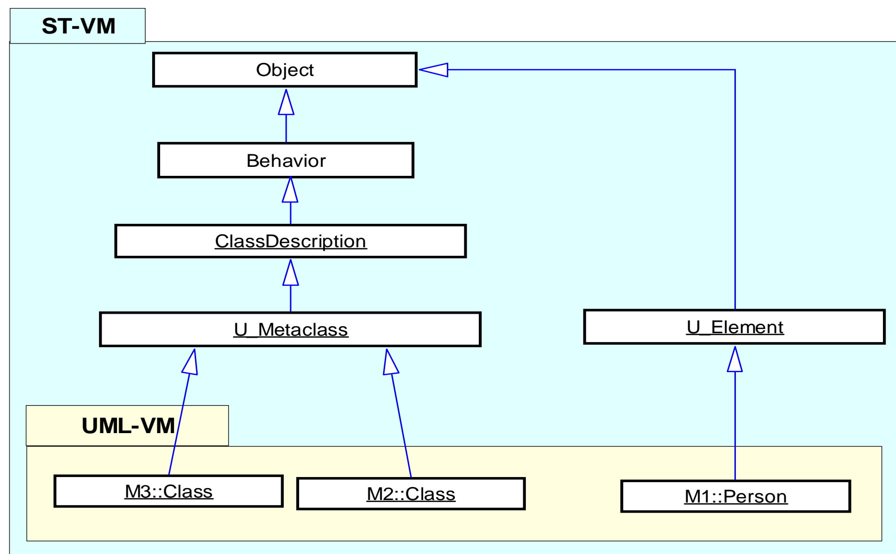
It is interesting to see how the object M1::Person was created. Here is the Smalltalk code for the createM1::Person-method in class Tester:[1]

```
builder := U::ClassBuilder new.
    builder.metaclass(M2::Class);
    builder.superclass(U::Element);
    builder.className('M1::Person');
    builder.instVarString( 'name age ');
    newClass := builder.createNewSubclass().
    self compileMethodsForM1::Person(newClass.)
    ^newClass.
```

1.I have qualified names with the layer name and used a double colon (::) as a separator. Smalltalk programmers will know that this is illegal Smalltalk syntax. I actually use underscore (_) in the code, but this is hard to read in a underlined name. I may later amuse myself with making a modified compiler that accepts the double colon.

The code illustrates that this is really quite simple. The U::Classbuilder collects information about the new class and then creates it in the createNewSubclass-method. The private method compileMethodsForM1::Person(newClass) in class Tester asks the new class for its compiler and then asks the compiler to translate the method definitions from textual to byte code form.

Figure 10: Class inheritance hierarchy of the first UML-VM.



The class hierarchy of this rudimentary UML-VM is shown in figure 10. It is interesting to note that both M3::Class and M2::Class are subclasses of U::Metaclass. This illustrates that the «instanceOf» and inheritance hierarchies are independent.

I expect that there will be substantial changes to this inheritance hierarchy when the UML-VM is extended with more UML concepts. It should probably reflect the class hierarchy of the definition document. (Even if this is not essential for the system semantics).

5. References

[U2P] Alkatel and others: Unified Modeling Language 2.0 Proposal. Version 0.651 (draft) http://www.u2-partners.org/artifacts/specs/U2P-UML2-v0.651-011213.zip 13 December 2001

[bluebook] Adele Goldberg, David Robson: Smalltalk-80. The Language and its Implementation. Addison-Wesley, Reading, 1983. ISBN 0-201-11371-6

[Ree01] Reenskaug: *Modeling Systems in UML 2.0 A Proposal for a Clarified Collaboration* Version of June 22, 2001. http://www.ifi.uio.no/~trygver/documents/2001/uml-20.pdf