Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In collaboration with Ecole des Mines de Nantes - France
2000

Transparent Strong Mobility using a Reflective Smalltalk

A Thesis submitted in partial fulfilment of the requirements
for the degree of Master of Sciences in Computer Science
(Thesis research conducted in the EMOOSE exchange project)
Master in Computer Science

By Gabriel Casarini

Promotor: Prof. Theo D'Hont (Vrije Universiteit Brussel)
Co-Promotor: Noury Bouraqadi et Thomas Ledoux (Ecole des Mines de Nantes)

Acknowledgements

I would like to thank all the people that in a way or other was involved in the development of my thesis.

Very special thanks to Noury Bouraqadi for his advice, time, motivation and providing new ideas all the time. Thanks also to Thomas Ledoux for arranging all the necessary things and providing the material to work.

I have a very big thanks for everyone in the Département Informatique at EMN. Thanks for being so kind and helping me when I needed it. Special thanks to Christine!

Very special thanks also to Annya Romanczuk for coordinating everything during the period of courses in Nantes and solving the infinite number of problems that appeared every week. Thanks also to Janick Le Hetet for addressing all the endeless administrative details.

Well… I want to be honest and say that living the whole year 14.000 km far away from home is really something and sometimes it was not easy! So I have very special thanks for everyone in the campus, specially for my students of the database course. Each of you made me feel at home in a way or other. Thanks for being so kind, polite and funny during my stay in France! Je vous remercie!

Of course, I have some special words for my family and friends in Argentina. Thanks for being so close with those wonderful letters and e-mails all the time. I really enjoyed every single comment and every single word you wrote!

In my endless list I also have very special thanks for the other EMOOSERs! Special thanks to Gabriela, Andrés and Xavier for being there when I needed help, advice or just some funny comment!

I honestly feel that all this was possible thanks to each and all of you.

Abstract

An object-oriented language is said to be *reflective* if it allows to handle and extend entities that are usually implicit. As a reflective language, Smalltalk provides access to many entities such classes and methods. However, its capabilities to control program execution are limited. In order to fix this weakness, an extension named MetaclassTalk, has been developed at EMN. The aim of this thesis is to explore the expressiveness of MetaclassTalk in order to achieve *Transparent Strong Mobility*. That is, allowing the migration of parts of a program from one node of the network to another one without stopping its execution. Thanks to the use of reflection, mobility can be introduced *transparently* without mixing application domain code with mobility code. This work lead to the design and implementation of a new version of MetaclassTalk on top of Squeak, a free Smalltalk written in itself.

Table of Contents

Introduction

Motivation

The central organizing principle of today's computer communication networks is Remote Procedure Calling (RPC). Conceived in the 1970s, RPC paradigm views computer-to-computer communication as enabling one computer to call procedures on the other. Each message sent across the network either requests or acknowledges a procedure's execution. This model is based on the notion of *mobility of control*, as conceptually, a *thread* of control originating at some network node continues execution at some other network node, and then comes back. Its simplicity and the exponential growth in size and performance of computer networks has undoubtly favor the success of RPC. In fact, this scheme of interaction is the basis on top of which the vast majority of distributed infrastructure for software applications is currently designed and implemented. However, there are some drawbacks in using RPC. The most noticeable one is that it requires a huge amount of network traffic every time remote applications interact. Another important drawback concerns the flexibility of the model itself to extend applications with new functionality—very often, different clients require certain level of customization of the computations performed remotely.

As an alternative, a different approach originates in the promising research area exploiting the notion of *mobile computation*. This paradigm views computer-to-computer communication as enabling one computer not only to call procedures in another, but also to supply the procedures to be executed. The idea is to package the all the remote interactions and dispatch them to a destination node where they can take place locally. Although there is a wide bunch of possibilities to achieved this goal, we are particularly interested in the case where the whole runtime image of a software component is transferred to another node, including its execution state. This is known as *strong mobility*. Thus, this model of interaction is based on the notion that a computation starting at some node of the network may eventually continue its execution at some other node.

There are many issues to consider when designing and implementing an infrastructure that provides strong mobility, like dealing with remote references, packaging the state of the process and so on. In fact, the underlying infrastructure that provides the facilities for strong mobility needs to have access to the internal execution environment of the components and applications that migrate. At some extent, this leads to a *mix of concerns*, as the traditional principle in software engineering is that a module should expose *only* its functionality through a public interface while hiding any detail about its internal implementation.

During the last decade, there have been several attempts to deal with the previous issues of accessing the internals of software systems. Apparently, allowing clients of software modules to control not only the interaction but also the implementation strategy of such modules, greatly increases adaptability and flexibility. And as a direct consequence, accessing the internal implementation greatly increases *reuse*. Among the several approaches that promote the access to the internal implementation and execution details of software systems, we mention *Open Implementation* [KdRB91] and *Separation of Concerns*. Their salient characteristic is the fact that they formally separate the base functionality provided by the module or application from the special purpose concerns such as synchronization, persistence and location control. Usually, the implementation of such concerns needs to observe and adjust the runtime execution of the modules. In other words, they need to *inspect* and *manipulate* the internal details of the system itself.

The particular domain of programs that are capable of describing and manipulating themselves has been under research for the last twenty years [Smi82, Mae87, Coi87, Riv96, BS99] and has been called *computational reflection*. Reflective systems have shown their practical interest for implemention because of their ability to represent the system, i.e. its functionalities and its implementation, within the system itself. Although it is a mature and clearly identified field in the research community, the concepts of reflection have almost had no influence on the development of new commercial programming languages—only very recently, some reflective features have been added to commercially available languages or products. A notable exception to all this is the Smalltalk language [GR83] which, from the begining, included reifications of the compilation process, methods, classes, and so on. In the last decade many attemps have been made to extend those features and extensions were introduced by systems like ClassTalk, NeoClasstalk and MetaclassTalk. All of them have contributed with mechanisms that bring more control over the execution of applications.

Having control over the execution of applications is the first step to take in the path of building an infrastructure that provides *strong mobility*. Thus our idea is to use reflective facilities and features as the means to introduce extensions in Smalltalk that give support for strong mobility. The challenge is to introduce those facilities in *transparent* way, enforcing the *separation of concerns*.

Organisation of the dissertation

The first part of our work, "Be Reflective", is about the advantages of applying reflection to extend the functionality provided by software systems. Chapter 1 introduces the concepts of reflection in general terms first; and in the context of object-orientation later. We analize the real advantages of reflective facilities and some of the approaches available that attempt to integrate it into systems. Chapter 2 presents the concepts and ideas behind the MetaclassTalk framework, an extension of Smalltalk that brings *behavioral reflection*. Chapter 3 is about our first contribution: the implementation of MetaclassTalk in Squeak—a free Smalltalk dialect written in itself. Many technical issues arised during the implementation that are discussed in this chapter. The choice of Squeak as our platform of development was mostly motivated because of its flexibility and the fact that it is freely available.

The second part of the work, "Be Mobile", is about mobile computation. Chapter 4 explores mobility in all its possible forms and approaches. We come to some conclusions and provide an analysis of requirements to develop a platform supporting *strong mobility*. Chapter 5 explores and discusses the reflective features available in Smalltalk to *enable* strong mobility. The objective is to analize the feasibility of implementing a platform providing facilities for strong mobility. Chapter 6 introduces our second contribution: some guidelines to design and implement a platform that enables strong mobility using reflection .We left the door open for an implementation in MetaclassTalk.

At the end we *summarize* the work, present *conclusions* and discuss different directions of *future work*.

Part I

Be Reflective

Chapter 1 Reflection

"Hence, [the machine] can, in particular, change the orders (since these are in memory!)—the very orders that control its actions".

John Von Neuman, 1958

"Reflective approaches appear to hold out the promise of dramatically changing the way that we think about, implement, and use programming languages and systems. There are those who fear, however, that by opening the door to unrestricted language level access to the unwashed masses, we are opening up a Pandora's Box".

[BF90]

1.1 Reflective Systems

In every-day life, reflection is about turning back the image of another object or the idea of examining one's own mental state, thinking about one's thoughts. In Computer Science, *reflection,* also known as *computational reflection*, is the domain of programs that describe and manipulate themselves. Based on some work made by Smith [Smi82] and Maes [Mae87] in the eighties, we provide the following definitions that help the understanding of computational reflective systems.

A *computational system* is a computer based system whose purpose is to answer questions about and/or support actions in some domain. The system incorporates *internal structures* to represent the different entities that define data and relations in the domain, and also a program prescribing how these data may be manipulated [Mae87]. A system is said to be *causally connected* to its domain if the internal structures and the domain they represent are linked in such a way that if one of them changes, this leads to a corresponding effect upon the other [Mae87]. A *reflective system* is a computational system in which the domain is the system itself. This means that both the system and its domain are causally connected. Reflective systems have the benefits that firstly, the internal representation always provides an accurate representation of the system and that, secondly, reflective systems can bring modifications to themselves by virtue of their own computations. Consequenty, reflective systems support both *self-inspection* and *self-adaptation*.

As an example of a causal connection consider a system steering a robot-arm, as shown in figure 1.1. The system incorporates structures representing the position of the arm. These structures may be causally connected to the position of the arm in such a way that *a)* if the robot-arm is moved by an external force, the structures change accordingly and *b)* if some of the structures are changed (by computation), the robot-arm moves to the corresponding position.

Notice that reflective computation does not contribute to solving problems in the external domain of a system; its benefits are not visible in the application domain itself. Rather, reflection contributes to the the internal organization of systems or to their interfaces to the external world, introducing new techniques that translate into more flexibility and adaptability [HVL95, Kic96, BS99].

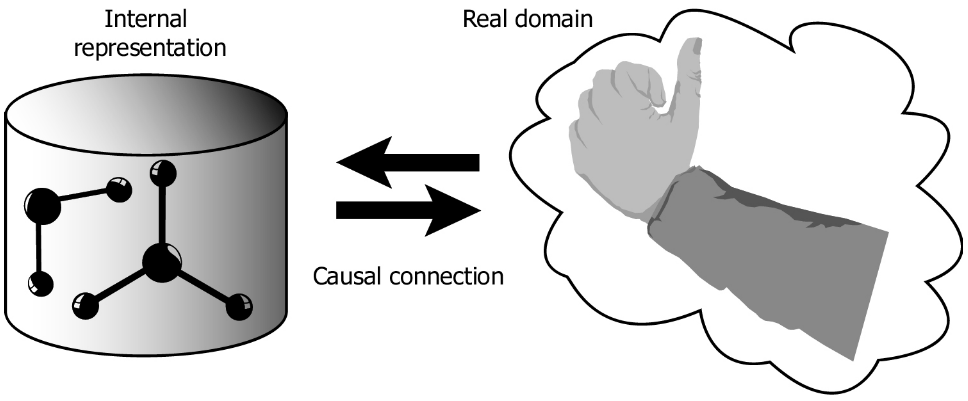


Figure 1.1 Causal connection between a domain and the internal representation provided by a system

1.2 Reflective Programming Languages

If we move to the area of programming languages, we need to deal with specific elements. This section attempts to provide some definitions and concepts on the matter.

Reflection is the ability of a program to manipulate as data something representing the state of the program during its execution as well as the infrastructure needed for the execution. Such manipulation involves two aspects: *introspection* and *intercession*. Both aspects require a mechanism for explicitly encoding the execution state as data [DBW93].

Introspection is the ability of a program to observe and therefore reason about its own state and interpretation [DBW93]. *Intercession* is the ability of a program to modify its own state of execution or alter its own interpretation or meaning [DBW93].

So *reflection* is the ability for a program to observe or change its own code as well as all aspects of its programming language (syntax, semantics or implementation), even at run-time [MJD96]. We can say that reflection implies the ability of any program to observe and modify the data structures actually used to run the program itself. A programming language is called *reflective* if it provides programs with reflection, that is, some special ability to observe and manipulate themselves. A programming language is said to have a *reflective architecture* if it recognises reflection as a fundamental programming concept and thus provides tools for handling reflective computation explicitly [Mae87].

According to des Rivieres, Kiczales, and Bobrow, *"a reflective language is a language allowing you to deal with explicit representations of implicit aspects of the language itself"* [KdRB91]. The following definition expresses the concept.

Reification is the mechanism used for explicitly encoding execution states and infrastructure as data [DBW93]. Reification is an operation by which something that was previously *implicit* or unexpressed is made *manifiest* and *explicitly* formulated for conceptual manipulation. A running program may want to reify such things as references to variables, source code to functions, interpreter to a given program, etc.

Reflection can be categorized in two major forms. On one hand, s*tructural reflection* is the ability of a language to provide a complete reification of both the program currently executed as well as a complete reification of its abstract data types [MJD96]. On the the other, *behavioral reflection* is the ability of a language to provide a complete reification of its own semantics and implementation (processor) as well as a complete reification of the data and implementation of the run-time system [MJD96]

It is important to remark that structural reflection does not refer to the structure of the abstract data types used by the program but rather to the internal elements of the language that represent the program itself. Behavioral reflection, on the other hand, refers to the posibility of manipulating the semanctics of a program, that is, the aspects involved during the execution. In the context of object oriented languages, structural reflection allows a developer to manipulate classes, methods, variables and so on. Behavioral reflection goes further and allows to alter the semantics of message sending, reception and method evaluation.

Structural reflection seems to be easier to implement and has been available for a long time in languages such as Lisp, Prolog and Smalltalk [MJD96]. Behavioral reflection, on the other hand, has not been so clearly tackled yet, essentially for performance reasons and also because it touches aspects governing the semantics of programs [MJD96]. Most implementations of languages with behavioral reflective features adopt interpretative techniques [Mae87, MJD96]. Interpreters ease modifications and react to them as soon as they occur, a remarkable advantage in reflection. But still more efficient implementations are needed. The combination of several approaches [MJD96] such as dynamic compilation and cache techniques can carry to important improvements in performance and thus make reflective laguages of real interest.

1.2.1 Reflective towers

As stated before, behavioral reflection allows a program to observe and modify the internal mechanisms involved during the execution. In most of the cases, the implementation of these kind of systems is based on interpretation techniques, as stated in [Mae87, MJD96]. This allows the program to reason about and modify its own interpreter. The interpreter itself can be seen as a program executed by another interpreter. Because this second interpreter belongs to the system, exactly the same conditions apply for it, that is, it can be observed and modified. In fact, it can also be seen as a program executed by a third interpreter which in turn is executed by a fourth interpreter and so on. In consequence, there is an infinite stack of interpreters. This stack of interpreters is known as *reflective tower* [Smi82].

Each interpreter of the reflective tower is at the same time [Smi82] *a)* a program executed by the interpreter of the level above and *b)* an interpreter for the execution of the interpreter in the level below. Each level can be clearly identified as a different *level of abstraction.* The first level, known as *base-level*, describes the program for the domain of the system. The second level is called *meta-level* and acts as interpreter for the basic-level. Moving up in the tower, each level is at the same time a base-level for the level above and meta-level for the level below. Figure 1.2 presents a representation of a reflective tower.

It is important to notice that each reflective system must find a way to actually implement the infinite tower using finite resources such as memory and time [Smi82]. The solution for this is to have a non-reflective interpreter at some level. This interpreter, called *default interpreter*, is written in another language, different from the language of the reflective system, and can be used to close the regression [Smi82]. So only a finite number of levels is needed to run the program; this number of levels is called the *degree of instrospection* [MJD96]. Eventually, the level of instrospection can vary at runtime if a different level of abstraction is chosen for the default interpreter.

In addition to the infinite reflective towers that concern the interpretation of reflective languages, there is the concept of *structural regression* [BS99]. This is the case of reflective class-based languages that effectively reify classes as first class objects [GR83, Coi87]. In such languages, each class is instance of a class, which in turn is instances of another class and so on. One solution for this is provided in the ObjVlisp system [Coi87] by introducing a metaclass, StandardClass, which is instance of itself.

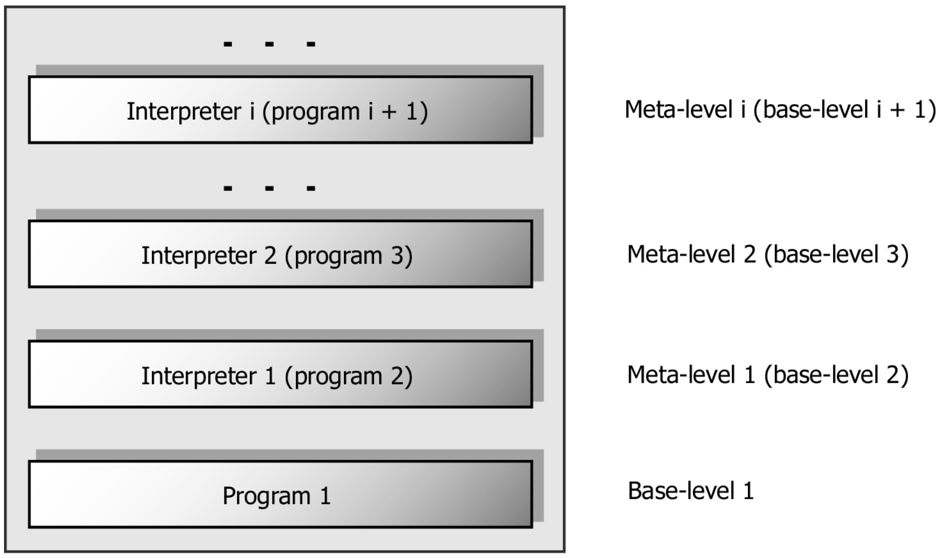

- - -
Interpreter i (program i + 1)
Meta-level i (base-level i + 1)
- - -
Interpreter 2 (program 3)
Meta-level 2 (base-level 3)
Interpreter 1 (program 2)
Meta-level 1 (base-level 2)
Program 1
Base-level 1

Figure 1.2. A reflective system has a pontentially infinite number of metalevels

The *base-level* of a system is the level at which the system reasons about its domain. At the base-level, a representation is provided to describe the entities of the domain, and their relationships. There is also a program to indicate how these representation should be manipulated. The *meta-level* is the level at which the system reasons about the base-level, providing explicit reifications of the internal representations and execution mechanisms. The meta-level is the level that provides the semantics of the base-level. Given a base-level, the meta-level itself has meta-level, which is then named meta-meta-level, and so on.

Notice that at the base-level, the fundamental structures and mechanisms that define how the language itself is implemented are reified and made explicit and manifiest at the meta-level. By nature, reflective languages are extensible; the modifications and extensions are controled and expressed at the meta-level. Figure 1.3 shows the meta-level and the base-level.

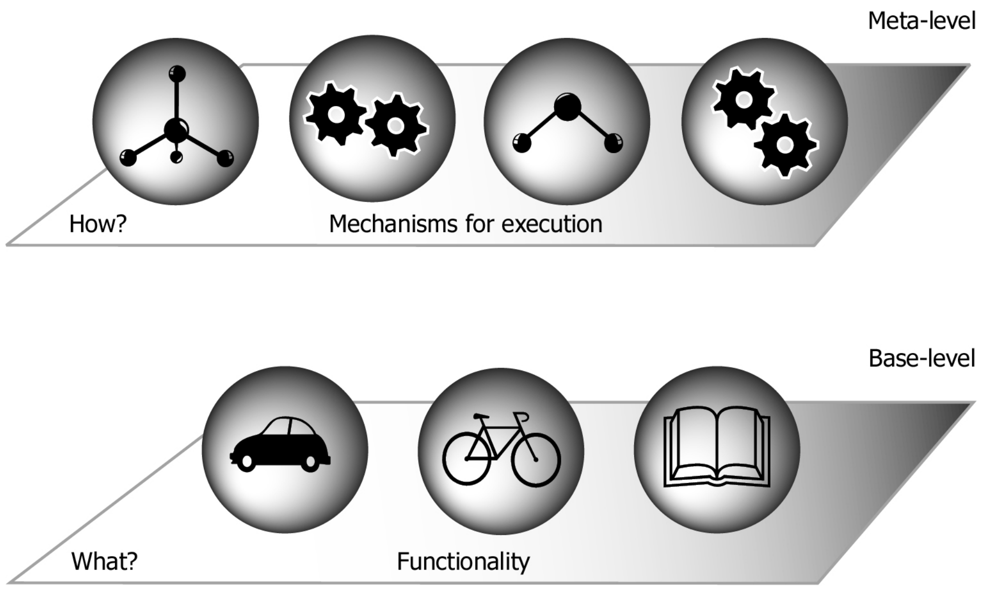


Figure 1.3 The base-level provides a representation of the domain; the meta level provides the execution mechanisms and internal representation of the entities of the base-level

1.3 Programming with Programmable Objects

Meta-programming is the art of developing methods and programs to read, manipulate, and/or write other programs. When what is developed are programs that can deal with themselves, we talk about *reflective programming*. In this section we provide some concepts related to meta-programming in the context of object oriented environments.

A *metaobject* is an object that controls one or more objects named *referents*. Metaobjects allow to control the structure and behavior of other objects. For example, a metaobject defines the way in which memory must be allocated for the structure of the object [BS99].

It can be said that the meta-level for any object is the level of discourse about the object; objects of the meta-level (metaobjects) are sentences or programs that are used to describe and manipulate objects at the original base-level. Metaobjects implement the non-algorithmic behavior of objects, i.e. they control the way in which these objects are executed. Pattie Maes states that *"the structures contained in the metaobject hold all the reflective information that is available about the object. The metaobject holds information about the implementation and interpretation of the object"* [Mae87].

Usually, each object has only one metaobject associated to it. Consider for example the systems 3-KRS [Mae87] and MetaclassTalk [BS99]. However, there are exceptions, as is the case of CodA [McA95], where each object can have many metaobjects associated to it. Figure 1.4 show the different cases.

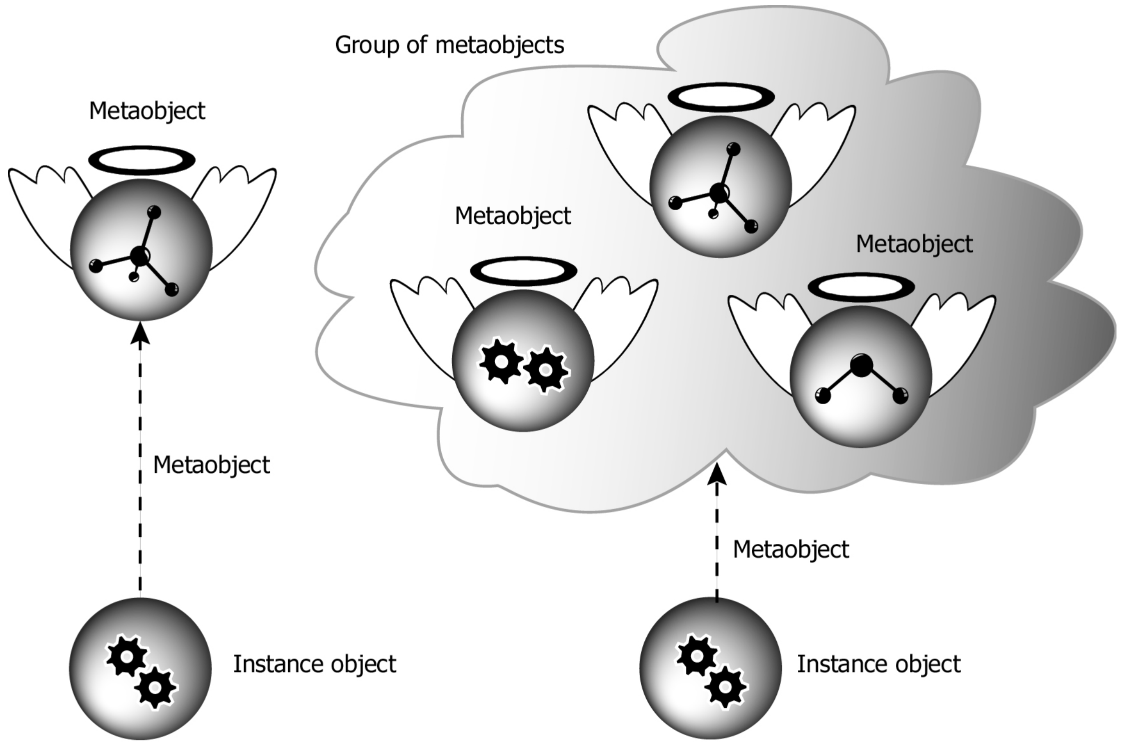


Figure 1.4 Instance object and metaobjects.

However, metaobjects are not only used to control the execution of instance objects. In some cases they are used to reified certain aspects or entities of the language itself like methods, classes and so on. Gregor Kickzales summarizes this approach in the following phrases: *"First, the basic elements of the programming language—classes, methods and generic functions—are made accesible as objects. Because these objects represent fragments of a program, they are given the special name of metaobjects. Second, individual decisions about the behavior of the language are encoded in a protocol operating on these metaobjects—a metaobject protocol. Third, for each kind of metaobject, a default class is created, which lays down the behavior of the default language in the form of methods in the protocol*" [KdRB91]

A good example of a language supporting metaobjects is Smalltalk, where specific classes are used to model meta-objects such as compiled methods, classes, messages, signals, exceptions, processes, contexts, blockclosures, method dictionaries, class organizers, etc. They also model associated tools for the programming environment such as the parser, the compiler, the browser, the class builder, the object memory ...

Meta-operations are operations that provide information about an object as opposed to information directly contained by the objects […] they permit things to be done that are normally not possible [LP90]. Related to this is the concept of *metaobject protocols* (MOPs) which are interfaces to the language that give users the ability to incrementally modify the language's behavior and implementation, as well as the ability to write programs within the language [KdRB91]. The idea behind metaobject protocols is to allow users to adjust design and implementation of languages to fit their particular needs. If handed properly, opening up the language design need not to compromise program portability or implementation efficiency. In a language based upon metaobject protocols, the language implementation itself is structured as an object-oriented program. This allows to exploit the power of object oriented programming techniques to make the language implementation adjustable and flexible [KdRB91].

1.4 What Are the Real Advantages of Using Reflection?

The traditional view in software engineering is to handle complexity by using abstraction by decomposition. In this way, implementation details are hidden from the user of a particular abstraction, and only a public interface of functionality is provided. The result is typically a *black box* approach to the development of application components which, it is argued, promotes re-use. The problem with this approach is that in practice it is often not possible or indeed desirable to hide all implementation details from the client (user) application. The fundamental problem is that by hiding implementation details, client programmers are forced to write extra code to fix incompatibilities and provide reasonable functionality, that would not be necessary if they had some control over the implementation of the modules involved. In other words, "*the client often best knows how the module should be implemented*"[Kic96]

The ideas and concepts of computational reflection have influenced the field of software engineering and development, moving the focus of research in the direction of more open and flexible systems as a way to tackle the previous problems. One approach comes from a new design principle called *open-implementation* [Kic96]. The idea is that clients are allowed to access, examine and manipulate crucial aspects of the modules involved in the application. An open implementation presents two interfaces: on one hand, there is a *primary interface* that provides the basic and clean functionality; on the other, there is a *meta-interface* or *adjustement interface* that can be used to accommodate the strategy decisions that underlie the primary interface. This model of dual-interface can be the basis for another paradigm named *separation of concerns* [HVL95]. This provides a more general form to formally separate the basic algorithm of the application from the different *concerns* involved in the execution and development, such as synchronization, persistence services, real-time constraints, location control, etc.

1.5 Summary

In this chapter we presented the concept of *computational reflection*. We also saw its advantages to open and make explicit the internal implementation of the systems.

Chapter 2 Concepts of MetaclassTalk

In Smalltalk [GR83], most of the entities used to describe the structure of the program are reified, particularly, classes, metaclasses and methods [Riv96]. These reifications make explicit the internal representation or structure of the entities used for programming, offering *structural reflection*. However, Smalltalk does provide little *behavioral reflection*, as there are just a few explicit reifications over dynamic aspects of the language, such as object behavior. Traditionally, *behavioral reflection* has not been so clearly tackled, essentially because it touches aspects governing the semantics of programs. This is particularly true in Smalltalk, where there are no facilities to control the interaction between objects.

Fortunately, some work has been done in order to provide Smalltalk with behavioral reflection based on the use of explicit metaclasses [Coi90, Riv97, BS99]. Systems like NeoClasstalk [Riv97] and MetaclassTalk [BS99] use *explicit metaclasses* and *classes as metaobjects* to provide more flexibility at level of the user. In this section, we will study the concepts around MetaclassTalk and its MOP.

2.1 Evolution Path: from Smalltalk to MetaclassTalk

This section provides a general overview of the reflective capabilities of Smalltalk and the way in which there enhanced and extended by ClassTalk, NeoClasstalk and MetaclassTalk. Figure 2.1 show the relations between the different systems.

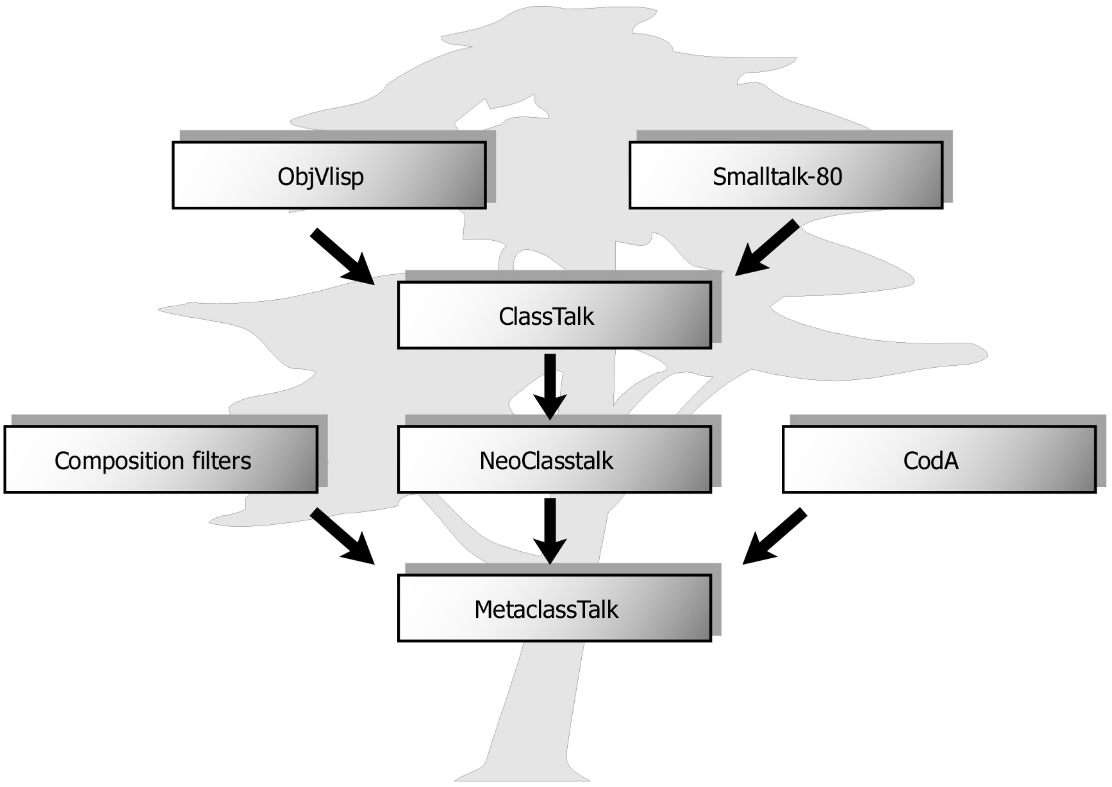


Figure. 2.1 Relations between the different systems

2.1.1 The Early Days: Smalltalk

Smalltalk [GR83] is a class-based object-oriented language that is almost entirely written in itself—Squeak might represent a notable exception, as it is completely written in itself. Smalltalk offers important advantages such as large portability, dynamicity, a fully unified world, graphical user interface builders, connection to database systems, development tools, etc. In fact, some people consider Smalltalk not only a language but rather a complete development environment. The implementation of Smalltalk itself is structured as an object-oriented program, expressed in Smalltalk and organized around meta-level objects that represent the classes, methods, lexical closures, processes, compilers and even the stack frames [rivard]. Although Smalltalk can be considered a reflective language, it is not considered a fully reflective language as it provides little behavioral reflection and no clear or explicit MOP to use it.

The following is a classification of the most important reflective aspects of Smalltalk [Riv96] that include meta-operation, structure, semantics, message sending and control state.

2.1.1.1 Meta-operations

As defined in early sections, meta-operations provide information about the object themselves, as opposed to the information contained by the object. In Smalltalk, the major meta-operations are defined in class Object, which is the the root of the inheritance tree. They allow us to:

- Handle the internal structure of objects: the methods instVarAt: and instVarAt:put: allow to read and write instance variables using and index of the name of the instance variable.

- Handle the object meta representation: there are two methods to deal with this. Method class answers the class of any object. Method changeClassToThatOf: changes the class of a given object and thus its behavior—notice that the latter method is not available in every implementation of Smalltalk or its name might vary among different implementations. There are some restrictions to the use of changeClassToThatOf and thus the change is not always possible: classes are required to have the same format, that is the same internal structure for their instances.

- Handle identity of objects: the message allOwners answers an array of all objects referencing the receiver of the message; identityhash returns an integer that identifies the receiver for an efficient access in dictionaries and become: to change the references to an object.

2.1.1.2 Structure: a Self-expressed Kernel

As stated previously, structural reflection implies the ability of the language to provide a complete reification both of the program and its abstract data types. Smalltalk is a unified language that only deals with objects. Each object is instance of a class that describes both the behavior and the structure of its instances. Classes as regular objects are in turn described by other classes called metaclasses (classes whose instances are classes themselves). Every metaclass has a single instance (except for metaclasses involved int the kernel of Smalltalk) and thus, a class/metaclass couple is established. Figure 2.2 defines the Smalltalk metaclass composition rule. This known as Smalltalk's *kernel*. Notice that the name of a metaclass is the concatenation of the name of its instance (a class) and the string 'class', since in Smalltalk, metaclasses are implicit—there are some exceptions to this like Behavior, ClassDescription, Class, Metaclass.

There are two metaclasses that describe the behavior of classes and metaclasses: Class and Metaclass respectively. This way, Object class inherits from Class. All metaclasses are instances of Metaclass. In order to stop the infinite structural regression, Metaclass class is also an instance of Metaclass. There are two abstract classes, Behavior and ClassDescription that regroup the common behavior between metaclasses and classes. Finally, notice that the major advantage of having a self-expressed kernel is that is can be extended to offer new functionaly at the level of the programming language. Systems like ClassTalk, NeoClasstalk and MetaclassTalk in fact provide an alternative kernel to extend Smalltalk.

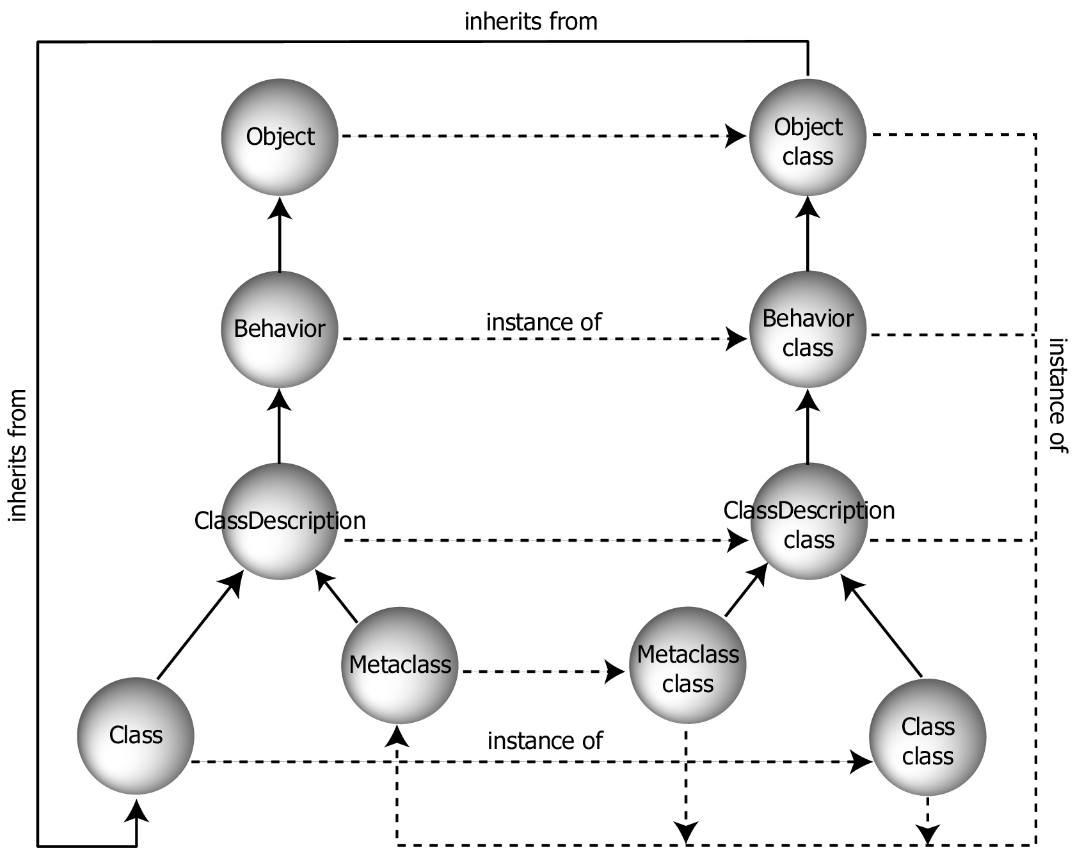


Figure 2.2. Smallatlk's kernel

2.1.1.3 Semantics

One of the salient features of Smalltalk is the fully reified compilation process. A compiler gives the semantics for the language and in the case of Smalltalk, as the compiler is available as regular objects, the semanctics are fully controlable [Riv96]. The compilation process is divided in to parts: parsing and code generation. Some of the classes involved in the process are: Parser, ProgramNode, CompiledMethod, Compiler, Decompiler. The way to extend the semantics is thus, by specializing or extending these classes.

2.1.1.4 Message Sending

The unique control structure of Smalltalk is message sending. It is composed of two phases: *method lookup* and *method application*. Method lookup concerns the search for the method to apply according to the receiver of the message sending. This happens at run-time and uses class information. Although it is not described in the language for reasons of efficiency, the necessary information for method lookup is accesible and modifiable from the language, as it all lies in classes. In this way, classes provide the following information: a dictionary with the compiled methods and the inheritance link (superclass instance variable). Method application, on the other hand, refers to the evaluation of compiled method in the context of the receiver. Actually, method application is performed by the virtual machine itself and is not available at the level of the user.

Notice that message sendings are not reified using instances of the message class except when the lookup fails. Only in that particular case, the doesNotUnderstand: method is sent by the VM to the original receiver with a reified message given as the argument. However, it is possible to call and explicit message send using the perform:withArguments: It is also possible to evaluate (apply) a compiled method by sending it the message valueWithReceiver:arguments:–notably, Squeak does not include this method!

2.1.1.5 Control State

Smalltalk is based on reified processes, and more generally on the objects needed to build a multiprocess system. There is one class, ProcessorScheduler, whose sole instance accessible through a global variable named Processor, coordinates the use of the physical processor by all processes requiring services. Processes, instances of class Process, may be suspended, resumed or teminated (using respectively suspend, resume and terminate methods). Semaphore class provides synchronized communication between processes with methods like wait and signal.

The BlockClosure class represents lexical closures. It freezes in a *block*, a piece of code (along with its environment) so that it may be evaluated later on. Blocks can have temporaries and arguments and their evaluation is provided by primitives named value, value: and valueWithArguments. Blocks are involved in process creation. Every time a new process is to be created, its body is enclosed within a block and then the message fork is sent the block. Given that blocks may share and environment, this allows to share objects among different processes.

2.1.2 ClassTalk

ClassTalk [Coi90] is the implementation in Smalltalk of the ObjeVlisp model, that can be used as an experimental platform to study explicit metaclass programming. To provide this platform, Classtalk introduced two protocols in Smalltalk: one to deal with object creation and another to deal with the execution of messages.

In Smalltalk, every object (with a few exceptions of primitive objects such as numbers) is created by sending the message new or new: to its class. This methods allocate the space in the memory and answer an instance of the class. The values of the instance variables take the initial value nil. Although at this extent, object allocation is uniform, object initilization is not. The user needs to perform the initialization on a second step, after allocation. One solution is to combine allocation and initilization into a single message using keyword selectors that specify a keyword for the value of each instance variable. Consider for example Point >> x:y: The problem with such creation methods are in most cases specific to each class. Classtalk addresses this problem by introducing a polymorphic method create:. This method receives an array of values and answers an instance initialized with such values.

The second protocol highlights the use of method lookup and method application. These notions are in fact already available in Smalltalk, but they are not very explicit in conventional implementations. In order to exploit this protocol, the sender must: a) send the message lookup:

to the object's class, passing the selector of the message as an argument and b) send the message applyTo:with: to the result of that—a compiled method—passing the receiver and the arguments.

2.1.3 NeoClasstalk

NeoClasstalk is an extesion of Smalltalk based on explicit metaclasses. There are two major differences between ClassTalk and NeoClasstalk. On one hand, NeoClasstalk introduces classes as metaobjects that control the application of methods on the instances. This is based on the use of method wrappers [BFJR98]. On the other, in NeoClasstalk objects have a dynamic internal structure, as opposed to traditional Smalltalk objects, where the internal structure is fixed in size. As a consequence, it is possible to delay the allocation of instance variables at run-time and change the class of an object, as structure compatibility is no longer an issue.

2.1.4 MetaclassTalk

MetaclassTalk is another extension of Smalltalk that also provides reflective facilities. It goes beyond NeoClasstalk allowing to control the access to internal structure of objects and enabling a finner control of the interaction between objects. Such control is made available to the user in the form of hook points represented by methods in the MetaclassTalk MOP [BS99]. One salient characteristic of MetaclassTalk, inherited from NeoClasstalk, is the use of classes as metaobjects.

2.1.4.1 Using Classes as Metaobjects

There has been some discussion in the past about the convenience or not of using classes as metaobjects of their instances [BS99]. Both of them, classes and metaobjects, play complementary roles in the architecture of an application. Classes provide the static description of objects (their structure and behavior), while metaobjects give the run-time semantics, providing the execution mechanisms. So at first sight, it seems very tempting to merge them together in only one entity, allowing classes to be also metaobjects. The following anaylsis compares the *pros and cons* of this.

2.1.4.2 Why Classes Should Not Be Metaobjects that Control the Execution of Instance Objects

The merging of classes and metaobjects has two main disadvantages. On one hand, the use of classes as metaobjects implies that all instances of the same class share the same metaobject. In this way, it is not possible to define different metaobjects for each instance of the same class. This point is particularly important as metaobjects cannot keep information or provide specific controlling mechanisms based on identity of the objects. On the other hand, if classes are metaobjects it is not possible that instances of different classes share the same metaobject.

The previous arguments come in favor of separating class responsibilities from metaobject responsibilities in two different entities.

2.1.4.3 Why Classes Should Be Metaobjects that Control the Execution of Instance Objects

The complementary roles played by classes and metaobjects come in favor of merging them together. Under this view, the control of the execution (metaobject's responsibility) could be linked to the description of the object (provided by its class). In fact, metaobjects control the object execution based on the information provided by classes (method definitions and internal object representation). On the other hand, if classes are metaobjects, it is possible to overcome the previous inconveniences by using inheritance and delegation to provide the desired functionality [BS99].

Summarizing, we conclude that there are no limitations if we choose to use classes as metaobjects. The main advantage of this approach is that we avoid incompatibilities between classes and metaobjects.

2.2 MetaclassTalk: the Language

2.2.1 Reified Entities

MetaclassTalk is an extension of Smalltalk with extra reflective facilities on top of explicit metaclasses. It allows to control the access to internal structure of objects and enables a finner control of the interaction between objects. This control is made available to the user in the form of hook points represented by methods in the MetaclassTalk MOP.

With the noticeable exception of variables, Smalltalk reifies almost all the entities used to describe the structure of a program: classes, metaclasses, methods and so on. These reifications constitute the basis on top of which MetaclassTalk was built. More particularly, classes are used as metaobjects for their instances. Metaclasses, on the other hand, are used to attach properties to classes [BS99] such asynchronous message sending, multiple inheritance, etc.

Being metaobjects, classes also control the creation process, the access to the internal structure and the execution of methods. As a class can have several instances, the same class can be the metaobject of several instance objects at the same time. The link between an object and its metaobject is created automatically when the object is instantiated. This way, the connection between an object and its metaobject is implicit.

The ability to control the method evaluation process is based on the use of methods wrappers—this was inherited from NeoClasstalk. This facility defines MetaclassTalk as a language that provides behavioral reflection.

2.2.2 Shifting from the Base-level up to the Meta-level

As stated previously, there is and implicit connection between and object and its metaobject (class) which is defined when the object is created. In order for metaobjects to control the execution of their referents, a control transfer is necessary from the base-level to the meta-level. This control shift can happen explicitly or implicitly.

2.2.2.1 Shifting Explicitly Up to the Meta-level

As Smalltalk, MetaclassTalk offers the possibility of moving explicitly up to the meta-level every time a metaobject is sent a message. As an example of an explicit shift to the meta-level, consider the creation of a new object. This is accomplished by sending the new message to a class. In this case, the explicit message sending to the class produces the transfer of control.

2.2.2.2 Shifting Implicitly Up to the Meta-level

Implicit shifting ocurrs when one object sends a message to another object. In MetaclassTalk, interaction among two objects is implicitly controlled at the meta-level by classes. This means that each attemp to access the internal state of an object or each message sending, is in fact controlled by some interactions between the metaobjects of the objects involved. Although this meta-level interaction takes place implicitly, it is defined explicitly in the metaclasses and is provided to the user in the form of hook points represented by methods in the MetaclassTalk MOP. Thus a mechanism is provided to customize the execution of the application.

In general terms, the idea is that every time a message is sent, the metaobject of the sender intercepts the message sending and delivers the message to the metaobject of the receiver, which in turn, decides how to handle the reception—the default action is to send the message to the receiver instance.

Figure 2.3 despicts the situation, in which an object s sends a message to another object r. When this happens, s's metaobject, say meta-s, takes control of the sending (1). This corresponds to an implicit passage from base level to meta-level. Then, meta-s gives control to meta-r (2). At this stage, meta-s retrieves the corresponding method for the message received (3) and finally, meta-s executes the method, giving control to **s**, at the basic level (4)
It's important to notice that the reification of step (3) leaves space to specialize the technique used for method retrieval, allowing for example the use of multiple inheritance. On the other hand, the reification of the method evaluation at step (4), allows to control the interpretation of the method and the returned value.

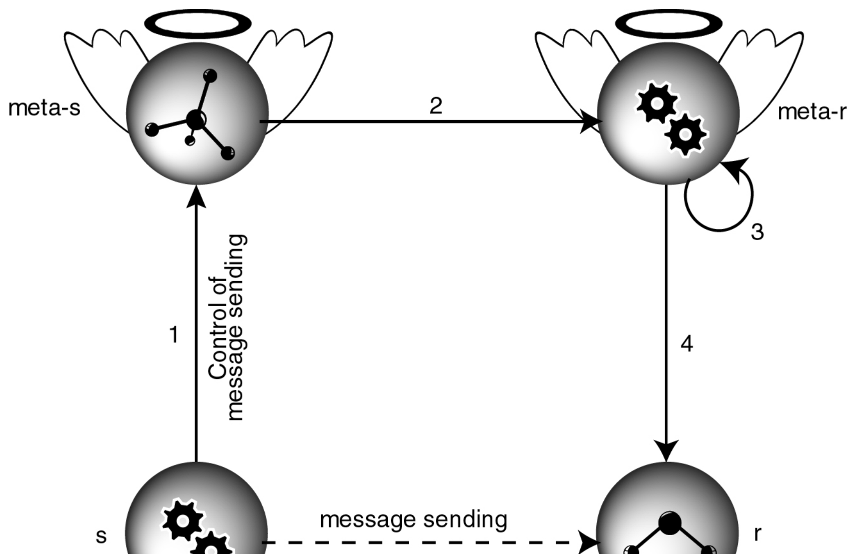


Figure 2.3 Metaobjects control the execution of instance objects

2.2.3 Infinite Regressions

As stated above, MetaclassTalk includes all the reifications available in Smalltalk. In this way, each class is instance of a metaclass—the class of a class—and in turn, metaclasses are instances of other objects, meta-meta-classes and so on. We explained before how these infite regresion is addressed in Smalltalk. In MetaclassTalk, the solution provided is based on the ObjVlisp's object model: there is one metaclass, called StandardClass, which is instance of itself, as shown in figure 2.4. This effectively closes the structural infinite regression, but we still need to deal with the infinite regression for behavioral reflection, that is, ensure the termination of the execution. This problem was presented in detail in section 1.2.1 as the *infinite towers of regression*.

By definition, StandardClass is its own metaobject—instance of itself—and thus, it controls its own execution. This produces an infinite loop of self meta-invocations that must be avoided in order to end the execution. The solution to this consists in delegating the execution of StandardClass's message sending on the Smalltalk virtual machine, which represent the default interpreter. In other terms, execution of the messages sent by a self-instanciated class is delegated to the virtual machine.

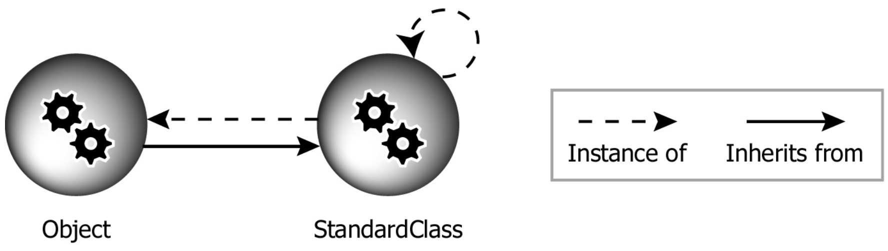


Figure 2.4 StandardClass is the kernel on top of which ObjVlisp was created

2.3 The MOP of MetaclassTalk

The idea behind metaobject protocols is to allow users to adjust design and implementation of languages to fit their particular needs [KdRB91]. In this section we present a brief description of MetaclassTalk's MOP. There are two categories of protocols available in MetaclassTalk: *a)* Methods that deal with the internal representation and access of the internal structure of objects and *b)* Methods that control interaction between objects. Table 2.1 summarizes the MOP.

Allocate
Allocates space in the memory for the header and body of a new object. Retruns a new instance object

New
Allocates and initializes a new instance object. Returns a new instance object

atIV: name of: anObject put: aValue	
Controls the access for writing of the internal structure of an object. Returns the value of instance variable named name	
Name	name of the instance variable
AnObject	object to be accessed
Avalue	the new value for the variable

atIV: name of: anObject	
Controls the access for reading of the internal structure of an object. Returns the current value of the variable	
Name	name of the instance variable
AnObject	object to be accessed

send: selector from: sender to: receiver arguments: args superSend: superFlag originClass: originCl	
Process the message sending. Returns the result after the method evaluation on the receiver	
Selector	selector of the message that was sent
Sender	object that sent the message
Receiver	object that will receive the message
Args	arguments of the message sent (a collection)
SuperFlag	a boolean value that indicates with true that the message was sent to super
OriginCl	initial class to start the method lookup

receive: selector from: sender to: receiver arguments: args superSend: superFlag originClass: originCl	
Intercepts the message reception to control the evaluation of the method down in the receiver (instance). Returns the result after the method evaluation on the receiver	
selector	selector of the message that was sent
sender	instance object that sent the message
receiver	instance object that will receive the message
args	arguments of the message (a collection)
superFlag	a boolean value that indicates with true that the message was sent to super
originCl	initial class to start the method lookup

lookupFor: selector arguments: args superSend: superFlag originClass: originCl	
Searches for the method to execute for the message. Returns the method corresponding to the selector	
selector	selector of the methos to look for
args	arguments of the message
superFlag	a boolean value that indicates with true if the search should start in the superclass
originCl	initial class to start the method lookup

apply: cm to: receiver arguments: arguments	
Performs the evaluation of the method. Returns the result of the evaluation of the method	
cm	arguments for the evaluation
receiver	instance object on which to evaluate the method
arguments	arguments of the message

Table 2.1 MetaclassTalk's MOP.

2.3.1 Controlling the Internal Structure of Objects

MetaclassTalk objects have the same structure as NeoClasstalk objects. This way, the structure of an object consists of two parts: the *header* and the *body*. The *header* corresponds to a field that keeps the reference to the class. The *body* corresponds to the fields where the internal state of the object is allocated. The internal state corresponds to the instance variables. Figure 2.5 shows the internal representation of an object.

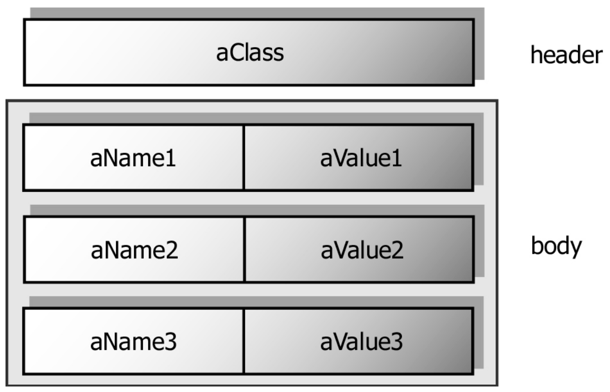


Figure 2.5 Internal structure of an instance object in MetaclassTalk

As oppossed to normal Smalltalk objects, in which the internal structure is fixed and calculated at creation time, the structure of MetaclassTalk objects is flexible and can vary dynamically in size, allowing to allocate and de-allocate instance variables at run-time. So the heading is the only part of the object that is fixed during the complete lifecycle of the object. The body, on the other hand, is allocated dynamically, in a lazzy fashion. Its fields are allocated the first time a reading or writing operation attemps to access a variable that was not read or wrote before.

In Smalltalk, each instance variable is always allocated in the same position of the internal structure; in MetaclassTalk, however, the body is implemented using a dictionary. There is one entry in the dictionary for each instance variable; each entry associating the name of the variable to its value. This approach allows to change at run-time the class of an object without the issue of structural compatibility [Riv96] that arises when changing the class of a normal Smalltalk object.

2.3.1.1 Creating New Objects

The creation of new objects in MetaclassTalk is done by sending the new message to a class, as in Smalltalk. The creation takes place in two phases: the *allocation* and the *initialization*, as in the ClassTalk object model.

The *allocation* consists in setting some space in memory for the heading of the object. As stated before, the allocation of the body takes place at run-time, under demand. The second stage of the creation is the *initialization*, by sending the message initialize to the new instance. This method is defined in the Object class and by default it does nothing; however, it can be specialized by the subclasses.

2.3.1.2 Accessing the Internal Structure

The access to the instance variables goes through two methods: atIV:of: and atIV:of:put:, that control the access for reading and writing operations respectively. These methods are automatically invoked on the metaobjects every time an attemp is made to access instance variables on the base-level objects. If an attemp is made to access a variable that is not currently in the body, then, new space is allocated in the body for that variable.

Writing

```
StandardClass >>
atIV: variableName of: anObject put: aValue

^( anObject bodyDictionary ) at: aSymbol put: aValue
```

The message bodyDictionary returns the body of anObject. The first thing to do is to check if indeed variableName exists in the body. If not, automatically new space is allocated for it—this the case when variableName is accessed for the first time. Then, the value aValue is stored in the memory field. The answer of the method evaluation is the new value of the instance variable, which is aValue in this case—notice that the value stored for the variable might not be aValue if the user customizes this hook point method.

Reading

```
StandardClass >>
atIV: variableName of: anObject

| values defaultValue |
values := anObject bodyDictionary.
^( values at: aSymbol
        ifAbsent: [ defaultValue := "retrieve default value for variableName".
                self atIV: variableName of: anObject put: defaultValue ] )
```

This method first checks if indeed variableName exists in the body. If not, automatically new space is allocated for it—this the case when variableName is accessed for the first time—and a default value is stored in the field. Otherwise, the current value of the variable is retrieved. The answer of the method evaluation is the value of the variable.

In the case that the variable is not allocated in the body, the default value to be stored in the field is provided by the instance object itself when its metaobject sends a special message. Every MetaclassTalk class includes methods that return the default value of every instance variable declared within the class. This methods, known as *Default Instance Variable Value* methods (DIVs) are automatically generated when the class is installed or recompiled. There is a special convention to assign the names of the DIV methods based on the name of the instance variables. Customizing the source code of the DIVs, the programmer can configure default values of the instance variables. This functionality, provided first in NeoClasstalk, is not available in conventional Smalltalk programming.

2.3.2 Controlling Behavior and Interaction Between Objects

In order to control the behavior of the objects, we need to control their interactions, that is, their communications. In MetaclassTalk interactions between two objects are decomposed in four stages: *message sending, message reception, method retrieval (lookup)* and *method application*. Each of these stages has been reified in the form of a hook point (method) that may be customized to provide other policies to control the behavior of their instances. The default policies provide exactly the same semantics for method execution that are already available in Smalltalk. Figure 2.6 shows a scenario of interaction to describe how the sending of a message from one instance object to another, is processed at the meta-level.

2.3.2.1 Controlling the Sending of Messages

Message sending is the first stage of object-interaction. Every time an instance object sends a message to another instance object, the completion of the sending is controlled by its metaobject. At the moment of the sending, there is an implicit shift from the basic-level to the meta-level which happens when the system intercepts the message and delivers the control to the metaobject of the sender. The metaobject, in turn, decides how to handle the *sending*. The default policy is to delegate on the metaobject of the receiver of the message, sending the message with any necessary parameters. The metaobject of the receiver decides how to handle the *message reception*. This default policy of delegation might be customized with subclasses. The following fragment of code is the hook point to control message sending. This menthod is invoked upon a message sending. Notice the delegation of control on the metaobject of the receiver.

```
StandardClass >>
send: selector from: sender to: receiver arguments: args superSend: superSend originClass: originCl

        ^receiver metaobject
                receive: selector
                from: sender
                to: receiver
                arguments: args
                superSend: superSend
                originClass: originCl
```

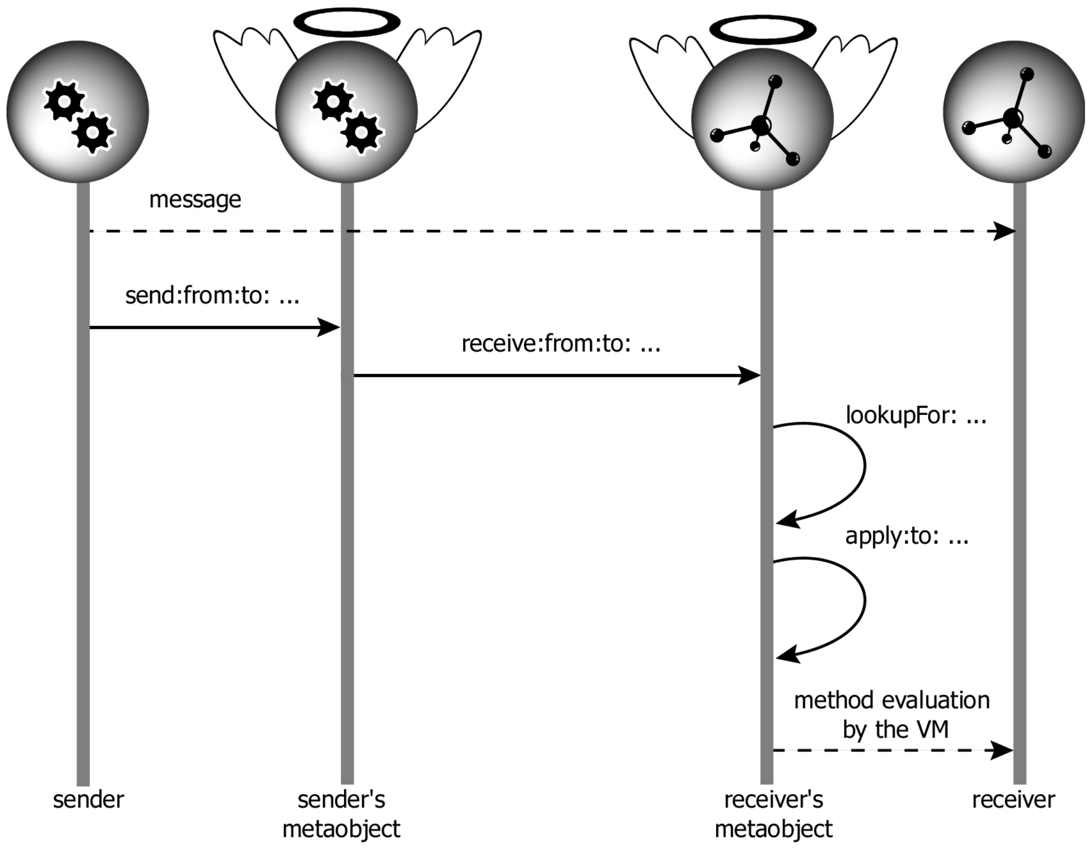


Figure 2.6. Scenario describing the sequence of messages sent at the meta-level

2.3.2.2 Controlling the Reception of Messages

As in the previous case, the reception of a message is controlled by a metaobject. In this case, the metaobject of the receiver accepts the delegation coming from the metaobject of the sender to handle the reception The default policy is to lookup for the executable code corresponding to the message and evaluate that code (CompiledMethod) on the receiver, with all the arguments of the message. When the evaluation (also known as interpretation) takes place, there is an implicit shift of control which by default is delegated to the virtual machine (primitive call). The following fragment provides the hook point to control message reception.

```
StandardClass >>
receive: selector from: sender to: receiver arguments: args superSend: superFlag originClass: originCl

    | cm arguments |
    cm := self lookupFor: selector arguments: args superSend: superFlag originClass: originCl.

    cm == nil
        ifTrue: [ cm := "Compiled method for #doesNotUnderstand".
                  arguments := "collection with the message that was not found" ]
        ifFalse: [ arguments := args ].

    ^self apply: cm to: receiver arguments: arguments
```

Two stages can be clearly identified in the fragment: *a)* Search for the executable code to be run (method lookup) and *b)* Execute the code in the context of the the receiver (object instance) with all the arguments provided with the message. The first step, method lookup, involves interaction with the class of the receiver, as it keeps the definition of the methods (MethodDictionary). Notice that the class is the metaobject itself.

If the executable code for the message is not found, an error should be reported. This is achieved by throwing an error message using the conventional mechanisms already available in Smalltalk (doesNotUnderstant protocol). Notice that all this default behavior can be customized with subclasses to provide, for example, pre/post condition before/after method evaluation.

2.3.2.3 Controlling the Method Lookup

Method lookup is the third step made explicit in MetaclassTalk to control the interaction, and thus, behavior of instance objects. Method lookup concerns the retrieval of the executable code to be evaluated. In Smalltalk, executable code is reified in the form of CompiledMethods stored in MethodDictionaries held by classes. In order to retrieve a CompiledMethod, three parameters are necessary: *a)* the selector of the method; *b)* an initial class to start looking for and *c)* some information to indicate if the search begins in the class itself or in its superclass—this the case when a super invocation is performed [BS99]. The method lookup must propagate the search up to the superclasses when the method is not available in the class where the search is started. The hook point is provided by the following method.

```
StandardClass >>
lookupFor: selector arguments: args superSend: superFlag originClass: originCl

    | method |
    superFlag
        ifTrue: [ method := "search the executable code in the superclass of originCl" ]
        ifFalse: [method := "earch the executable code in originCl" ]
    ^method
```

This default behavior might be customized with subclasses to support, for example, a model with multiple inheritance or a different policy of method retrieval.

2.3.2.4 Controlling Method Evaluation

This is the final step for the MetaclassTalk approach to control the objects behavior. This reification allows the programmer to control *where* and *when* the receiver object runs (interpretation), as well as its overall importance (e.g. priority) and independence. Having an explicit execution model also enables methods to be interpreted in different ways depending on the situation. For example, if we are debugging an object, we may wish to execute its methods on a special purpose debugging virtual machine or interpreter whereas normally methods are executed as native machine code. So the role of this reified aspect is to determine how and when to execute methods. The default policy provided by MetaclassTalk is to leave the interpretation to the default interpreter (underlying virtual machine).

The hook point is provided by the following method.

```
StandardClass >>
apply: cm to: receiver arguments: arguments

    | result |
    result := invoke the evaluation of cm with the arguments in the context of the receiver object.
    ^result
```

This hook point is the right place to customize some manipulation of the result of the interpretation, before delivering it back to the sender.

2.4 Using MetaclassTalk for Squeak

This section introduces the protocol to create new classes in MetaclassTalk for Squeak [IKMWK97]. Some examples are provided to complete the explanations.

2.4.1 Creating Classes and Metaclasses

The protocol to create new classes and metaclasses in Squeak using MetaclassTalk is different compared to the previous version. We prefered to keep the long Smalltalk tradition and introduce only minimal changes to the standard protocol and way of working. As it should be clear by now, MetaclassTalk is all about explicit metaclasses. So in order to create a new class or metaclass, a metaclass must be provided explicitly. The protocol of creation is the following:

```
aSuperclass subclass: #aClassName
        instanceVariableNames: aString
        category: anotherString
        metaclass: aMetaclass
```

The following example shows the declaration of a class MyFirstClass, with two instance variables, a and b, as a subclass of Object. MyFirstClass is explicitly declared as an instance of StandardClass.

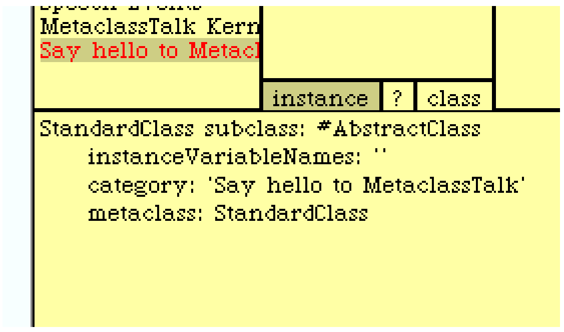
```
Object subclass: #MyFirstClass
        instanceVariableNames: 'a b'
        category: 'Say hello to MetaclassTalk'
        metaclass: StandardClass
```

The next example is a little more subtle: we want to work with abstract classes. We declare a new metaclass, AbstractClass, subclass and instance of StandardClass. AbstractClass is a metaclass (not a class) because it inherits from StandardClass. Abstract classes get never instantiated, thus we redefine the new method. Figure 2.7 shows the declaration in MetaclassTalk.

```
StandardClass subclass: #AbstractClass
        instanceVariableNames: ''
        category: 'Say hello to MetaclassTalk'
        metaclass: StandardClass

AbstractClass >>
new

^self error: 'You are trying to instantiate an abstract class!!'
```



```
StandardClass subclass: #AbstractClass
    instanceVariableNames: ''
    category: 'Say hello to MetaclassTalk'
    metaclass: StandardClass
```

Figure 2.7. The declaration of AbstractClass in MetaclassTalk for Squeak.

Now that the metaclass providing the behavior of abstract classes is installed, we proceed to create AbstractCollection as an instance of AbstractClass and subclass of Object. The fact that AbstractCollection inherits from Object makes it an class (not a metaclass).

```
Object subclass: #AbstractCollection
        instanceVariableNames: ''
        category: 'Say hello to MetaclassTalk'
        metaclass: AbstractClass
```

Any attemp to instantiate AbstractCollection by sending the new message, throws an error exception. Notice that the previous protocol is only necessary to work with MetaclassTalk.

Normal Smalltalk classes must be created using the conventional Smalltalk protocol—without explicitly specifying a metclass.

2.4.2 Adding, Editing and Removing Methods

Once a class or metaclass has been created, the procedures to add, edit and remove methods are exactly the same used in Squeak for conventional classes. Notice that when the *class* button is selected in a browser, what in fact is displayed in the PluggableList that shows the methods, are the instance methods declared in the metaclass of the class. This can be a bit strange at the beginning but encourages the adoption of the idea that classes are instances of other explicit objects. Class methods are no longer declared interacting with the *class side* of a class, but rather as instance methods of an explicit metaclass.

2.5 Summary

In this chapter we presented MetaclassTalk, which is an extension to Smalltalk that provides *behavioral reflection*. MetaclassTalk allows to *control inter-object communication*.

Chapter 3 MetaclassTalk for Squeak

"Squeak stands alone as a practical Smalltalk in which a researcher, professor, or motivated student can examine source code for every part of the system, including graphics primitives and the virtual machine itself, and make changes immediately and without needing to see or deal with any language other than Smalltalk"

[IKMWK97]

Squeak [IKMWK97] is an open, highly-portable implementation of Smalltalk whose virtual machine is written entirely in Smalltalk itself. The project was started in 1996 by the Smalltalk's original founding fathers Alan Kay and Dan Ingalls, as an answer to the search of a development environment for educational purposes that could be used—and even programmed—by non-technical people and children. Perhaps, the most noticeable characteristic of Squeak is that it involves a huge world-wide community of users that include researchers, programmers and professional developers that collaborate to make the system evolve. Current realeases are completely stable and run bit-identical images across a wide portability base [IKMWK97].

The original implementation of MetaclassTalk was built on top of the NeoClasstalk system running on VisualWorks 2.0. Our mission was to port MetaclassTalk to Squeak as a framework that deals with explicit metaclasses and extends Smalltalk with behavioral reflection. The idea was to remove from the previous implementation all the NeoClasstalk legacy code, providing a clean port of the MOP itself. The following sections explore the issues arised during of the implementation, discussing the differences with the previous version whenever possible.

3.1 Internal Structure of Objects and Metaobjects

As stated in the previous chapter, the internal structure of MetaclassTalk objects is based on the same structure that NeoClasstalk provides. In this way, every MetaclassTalk object is represented internally by two Smalltalk objects:

- a header: it is implicitly created by Smalltalk when the a new object is instanciated. It is used to keep a reference to the class and it is not available for manipulation at the meta-level.
- a body: it is used as a repository to keep the internal state of the object. It is inaccessible for the instance itself; rather, the body can only be manipulated at the meta-level by the metaobject of the instance. The body is implemented as a dictionary in which instance variables are allocated at run-time.

In the previous implementation of MetaclassTalk, the body was not kept inside the object itself but rather outside. A global variable, Accessors, kept a big dictionary that associated each instance object with its body. Accessors was manipulated at the meta-level, by metaobjects, when the internal state of an object must be accessed for reading or writing.

In our implementation, the approach is different. The body of an object is encapsulated within the object itself. This means that the object has a reference to a dictionary where all the instance variables are stored.

Although MetaclassTalk classes—which play the role of metaobjects—are, of course, objects, their internal structure must include some extra information required by the Smalltalk virtual machine:

- A class
- A reference to the superclass
- A method dictionary where the compiled methods of the instances of the class are stored
- The *format*, which is an integer value that encodes the internal structure of the instances, such as number of instance variables, and indicates if the instances are variable or fixed, an so on.
- A body variable to store some other variables normally available in any standard Smalltalk class, such as the name of the instance variables, a short comment about the class, etc. The body is in fact a dictionary that associates each variable name to its value.

In order to deal with this requirements, our implementation of MetaclassTalk classes and metaclasses—metaobjects—internally handles five instance variables: class, superclass, methodDict, format and body. The first one is implicit, the next three are explicit and the latter is our design choice to store the rest of the structure.

3.1.1 Object Format

Everytime an object is created, the system must allocate some space in the memory for the new object. The memory requirements of the instances are encoded within their classes using an integer value known as *format*. In other terms, the format is an integer that encodes the kinds and numbers of variables of instances of a class. This value is manipulated by low-level methods.

As the format specifies the internal structure of the objects, it highly dependends on the internal characteristics of each Smalltalk implementation and the techniques they use for allocation and manipulation of memory. This is the reason why the format message sent to the same class will answer different values in VisualWorks and Squeak.

In order to implement MetaClassTalk in Squeak, we need to know the values of the format. In the case of object instances of fixed size, as we need to allocate only one instance variable—for the body—the value of format is 132. In the case of metaobjects, four instance variables need to be allocated: superclass, methodDict, format and body–as class is implicit. And thus, the value of format for classes and metaclasses of fized size is 138.

Summarizing, the format message must return 132 when sent to a class and must return 138 when sent to a metaclass. However, in MetaclassTalk, we only deal with metaobjects, that play the role of a class or metaclass dependending on the kind of objects obtained when they get instanciated. As all metaobjects are instances of StandardClass or any of its subclasses, how do we determine what the value of format is when a new metaobject is created? How do we decide of the metaobject is a class or a metaclass? The answer comes after analyzing the metaobject's inheritance chain. If there is a path in the inheritance tree that goes from the metaobject to

StandardClass, then the metaobject is a metaclass, as it inherits all the behavior of metaclasses. Otherwise, the metaobject inherits from Object and it is class. This analysis is used to determine which value must be assigned for the format instance variable of a new metaobject.

3.1.2 Object Allocation

As stated previously, the creation of a new object implies two steps: memory allocation and object initialization. Creation of a MetaclassTalk object is done in the same way it is done in Smalltalk: by sending the new message to the class. Below we provide the definition of the methods involved:

```
StandardClass >>
new

        ^self allocate initialize



StandardClass >>
allocate

        | newObject |
        newObject := self basicNew.
        newObject bodyDictionary: (Dictionary new: self instSize).
        ^newObject
```

3.2 Object Interactions in a Heterogeneous Environment

MetaclassTalk extends Smalltalk with explicit metaclasses that provide behavioral reflection in the form of metaobjects. Up to now, we have not mention how MetaclassTalk metaobjects and their extended functionality are integrated within Smalltalk.

There two radically different approaches to make Smalltalk be MetaclassTalk compatible. On one hand, we could convert all the objects available in the Smalltalk image to MetaclassTalk objects. Under this approach, each object would have a metaobject, all metaclasses would be explicit and Smalltalk would be a truly reflective and homogeneous environment. However, the overall performance of the system would dramatically slow down because of the overhead introduced by MetaclassTalk's control of execution. On the other hand, we can make Smalltalk and MetaclassTalk objects co-exist and cooperate. Under this approach, MetaclassTalk's kernel would be integrated within Smalltalk. The main advantage of this is that there is no performance.

For this implementation, likewise the previous version, we prefered to integration of Metaclasstalk within Smalltalk. This integration raises the issue of interaction between normal Smalltalk objects and MetaclassTalk objects. There are exactly four cases of interaction:

- a Smalltalk object sending a message to another Smalltalk object
- a Smalltalk object sending a message to a MetaclassTalk object
- a MetaclassTalk object sending a message to a Smalltalk object
- a MetaclassTalk object sending a message to another MetaclassTalk object

Each of this cases must be treated in a different way. In order to deal with this, we base the explanation on some short fragments of Smalltalk code. Consider two classes, Person and MTPerson, both of them defining the method sayHelloTo: that receives another person as an argument:

```
Person >>
sayHelloTo: aPerson

^aPerson accept: 'Hello!' from: self

MTPerson >>
sayHelloTo: aPerson

^aPerson accept: 'Hello!' from: self
```

Although both of them are subclasses of Object they are deeply different. Person is a normal Smalltalk class, whose metaclass is implicit, and does not offer any control on the execution of the program. MTPerson, on the ohter hand, is an instance of StandardClass and thus it plays the role of metaobject for its instances. Thus it will control their interaction. The following sections discuss the different cases in detail.

3.2.1 A Smalltalk Object Sending a Message to Another Smalltalk Object

This the case when we evaluate the following from the transcript:

```
| p1 p2 |
p1 := Person new.
p2 := Person new.
p1 sayHelloTo: p2
```

In this case execution takes place in the normal way and all the interaction is handled by the underlying virtual machine. No explicit mechanisms of control are available for the message sending.

3.2.2 A Normal Smalltalk Object Sending a Message to a MetaclassTalk Object

This the case when we evaluate the following from the transcript:

```
| p1 p2 |
p1 := Person new.
p2 := MTPerson new.
p1 sayHelloTo: p2
```

In this case, the virtual machine handles the dispatch of the message from p1. However, as p2 is a MetaclassTalk object, its metaobject, MTPerson, should capture and control the reception of the message.

The technical solution to put p2's metaobject in the middle of the scene is based on the use of *method wrappers*. As discussed in [BFJR98], *"wrappers are mechanisms for introducing new behavior before and/or after, and even in lieu of, an existing method."*. Basically, this technique consists in replacing in the method dictionary the compiled method A generated after the compilation of the source, with another compiled method B that acts as a wrapper for A. Then, at runtime, B controls the evaluation of A, being able to perform some actions after and before A's evaluation.

It is important to notice that wrappers are not used in MetaclassTalk to introduce behavior, but rather to capture method invocation. They are used when methods are compiled to introduce in the method dictionary the necessary code that shifts the control up to the meta-level, sending all the parameters of the invocation. The inserted code is in fact made available by cloning pre-compiled methods called prototypes. The following is the source code of one of these prototypes. Notice that what it does is just delegate control on the metaobject by sending the message #receive:from:to:arguments:superSend:originClass:. More detailed information about this technique is available later in this chapter.

```
"Body of a prototype method"

methods := #(1).
name := #(2).
^self metaobject
        receive: name first
        from: (thisContext sender receiver)
        to: self
        arguments: #()
        superSend: false
        originClass: nil
```

Summarizing, when p1 sends the message accept:from: to p2, the code of the previously shown method gets evaluated, performing a shift of control to the meta-level (p2's metaobject). Notice that among the arguments of the invocation, we send the real compiled method for accept:from:, obtained by the expression methods first—the reference to the wrapper is stored in the literal section of the CompiledMethod.

3.2.3 A MetaclassTalk Object Sending a Message to a Normal Smalltalk Object

This the case when we evaluate the following from the transcript:

```
| p1 p2 |
p1 := Person new.
p2 := MTPerson new.
p2 sayHelloTo: p1
```

In this case we do not need to intercept the reception and invocation of accept:from: at p1's side because it is a Smalltalk object. However, we may need to do something when p2 attempts to send the message. This means that we must be able to intercept the sending of accept:from:

In this case, the technique used to achieve this is a kind of *program transformation* that reifies the sending and shifts the control to p2's metaobject. This program transformation is performed when sayHelloTo: is compiled. So internally, the compiled code (forget the wrappers for a moment) corresponds to the following source:

```
"Source code written by the user"

MTPerson >>
sayHelloTo: aPerson

^aPerson accept: 'Hello!' from: self
```

```
"Decompiled version"

MTPerson >>
sayHelloTo: aPerson

^self metaobject
        send: #accept:from:
        from: self
        to: aPerson
        arguments: (Array with: 'Hello!' with: self)
        superSend: false
        originClass: nil
```

The default implementation of send:from:to:… sends the receive:from:to:… to the metaobject of the receiver, which is Person in the previous example. This means that we need to define receive:from:to:… in the Smalltalk kernel—more specifically in class Behavior.

Summarizing, when p2 receives the message sayHelloTo: it will send a message to its metaobject which will take the appropiate actions—at some moment from the meta-level, the accept:from: message will effectively be sent to p1. There some references to the technique we applied here in [VB00]. Basically it consists in replacing the MessageNode obtained by the parser with another one that corresponds to the message sent to the meta-level. So this transformation ocurrs before the generation of the bytecodes.

3.2.4 A MetaclassTalk Object Sending a Message to Another MetaclassTalk Object

This is the case when we evaluate the following from the transcript:

```
| p1 p2 |
p1 := MTPerson new.
p2 := MTPerson new.
p1 sayHelloTo: p2
```

In this case, we may intercept everything: the sending of accept:from: performed by p1 and the reception of accept:from: by p2. The sending of accept:from: is captured by means of a *program transformation* that shift the control up the p1's metaobject by sending it the message send:from:to:… That method, in turn, sends the message receive:from:to:… to p2's metaobject, which decides how to control the reception. So in this case, there is a direct interaction between the metaobjects before p2 receives the message.

```
"Decompiled version"

MTPerson >>
sayHelloTo: aPerson

^self metaobject
        send: #accept:from:
        from: self
        to: aPerson
        arguments: (Array with: 'Hello!' with: self)
        superSend: false
        originClass: nil
```

3.3 Transforming and Wrapping Code

As explained before, we used different techniques in order to shift the control of execution up to the meta-level. We explained that the bytecode that effectively gets executed at runtime is not exactly the compiled version of the source code provided by the programmer, although they are related. In fact, the compilation process goes through two extra intermediate stages before the CompiledMethod is installed in the class. The first stage, applies some transformations to the source code in order to control the sending of messages during the evaluation of the method. The second stage builds a wrapper out of the compiled method to control its invocation. Figure 3.1 shows the process. The following two sections go into detail about the implementation of code transformation and method wrappers.

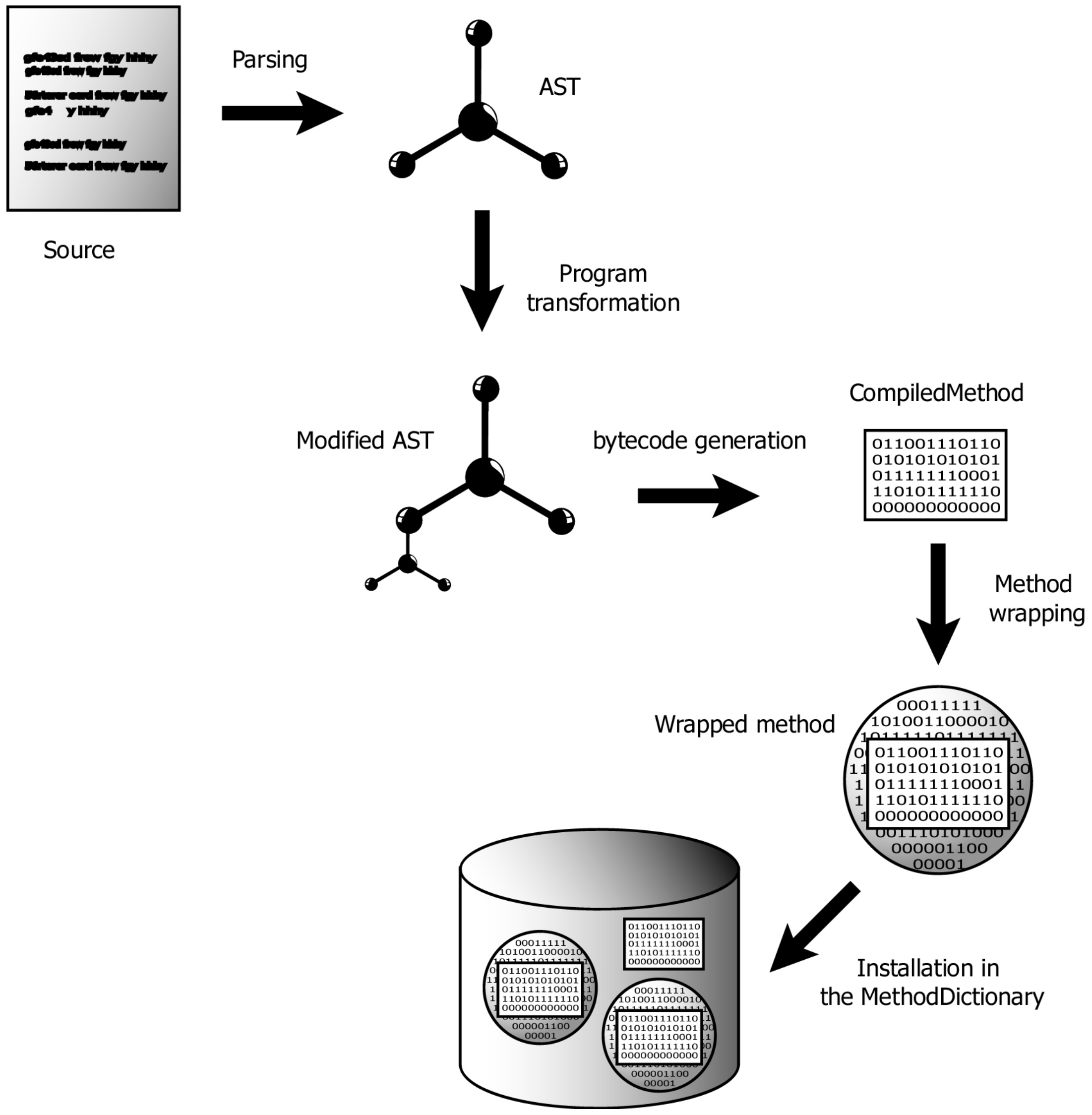


Figure 3.1. The different techniques used at the implementation

3.3.1 Implementing Methods Wrappers

As stated before, method wrappers are used to implement the reification of method invocation. Wrappers are cloned from already compiled methods called *prototype methods*. The execution of these methods cause the shift up to the meta-level by sending a message to the metaobject of the receiver. Prototypes are indentified by the following selectors and installed as instance methods at CompiledMethod class:

```
protoMethod
protoMethodWith:
protoMethodWith:with:
...
```

When the compiler needs to *wrap* a method, it builds on the fly a selector according to the number of parameters of the method, retrieves the corresponding prototype method and makes a clone of it. For example, in the case of wrapping sayHelloTo: the wrapper to use is protoMethodWith: and in the case of accept:from: the wrapper is protoMethodWith:with:. Then, the clone is pointed at the real method (already compiled) and installed in the MethodDictionary of the class. At runtime, method lookup returns the clone, allowing to achieve the jump to the meta-level in a completely transparent way.

The following fragment is the source code of a prototype method. The declaration of the temporary variables is just to reserve space in the literal frame of the compiled code, in such a way that later, the reference to the wrapped method and the selector of the method can be stored. In the previous implementation—VisualWorks—there was no need to store the selector within the compiled method; however, it was necessary to do it in Squeak, as there is no other way to determine which is the selector associated to a CompiledMethod.

```
protoMethodWith: arg1

| methods name |
methods := #(1).
name := #(2).
^self metaobject
        receive: name first
        from: (thisContext sender receiver)
        to: self
        arguments: (Array with: arg1)
        superSend: false
        originClass: nil
```

This implementation of MetaclassTalk followed the same approach of the previous version to implement prototype methods. Thus, the clones were installed as instances of CompiledMethod. However, in many cases it would be useful to override the default behavior of the protypes. Consider for example the case of browsing the source code of a class that includes wrappers within it methods. In that case, what we want to display in the browser is not the source of the *wrapper* but rather the source of the *wrapped* method (referenced from the *wrapper*). More specifically, we need to specialize the #decompileString method.

The obvious solution for this is to introduce a subclass of CompiledMethod, say WrapperMethod, that overrides the inherited behavior where necessary. Then, we install the prototypes as instances of this class. Following the example, when WrapperMethod is sent the message #decompileString, it will delegate the responsibility to the CompiledMethod it points at—the wrapped method compiled with the source code provided by the programmer. In this way, the management of wrapping/unwrapping methods is completely transparent from the point of view of the browser (or any object requiring the source code).

3.3.1.1 Un-wrapping Methods

When a metaobject is to perform the application—execution—of the CompiledMethod it has just retrieved from a MethodDictionary, if first needs to unwrap the method. This is because what needs to be applied is not the method itself but rather the wrapped method it points at—which is the compiled version of the code typed by the programmer. The following fragment shows this explicitly:

```
StandardClass >>
apply: cm to: receiver arguments: args

^(self removeWrapper: cm) valueWithReceiver: receiver arguments: args
```

3.3.2 About the Implementation of Program Transformation

As stated by Fred Rivard, Smalltalk compilation uses the existing Smalltalk code for its own needs, and is designed as a regular object-oriented program which is causally connected to the language. Thus, using Smalltalk code, we can extend its compilation process [Riv96].

In our case, we used these facilities to implement the reification of message sendings, based on transformation of the source code before the compilation. Program transformation consists in introducing changes to the AST built from the source code provided by the programmer. Consider again the example shown before. The following code corresponds to the source provided by the programmer:

```
"Source code written by the user"

MTPerson >>
sayHelloTo: aPerson

        aPerson accept: 'Hello!' from: self
```

Before compilation, the method is transformed into this:

```
"Decompiled version"

MTPerson >>
sayHelloTo: aPerson

self metaobject
        send: #accept:from:
        from: self
        to: aPerson
        arguments: (Array with: 'Hello!' with: self)
        superSend: false
        originClass: nil
```

The compilation process itself is divided in to stages: parsing and code generation. In our approach, we introduce the changes during the parsing. The parsing consists in finding the structure behind the source and results in a parse tree that gives the compiler the meaning of the source code. As expected, Smalltalk reifies both the parsing process and the data structures used for the parsing. However, as the classes involved during the parsing are rather different in VisualWorks and Squeak, our implementation of the programe transformation in MetaclassTalk is quite different from the previous implementation.

Technically, the transformation introduces some extra nodes in the tree that represent the structure of the modified version of the source code. In order to localize the changes of the parsing process, we introduced some new classes in the system: MTParser and MTVariableNode. It was also necessary to add and extend methods in some of the standard Smalltalk classes involved in the process, like MessageNode, Encoder and Compiler.

3.3.2.1 Program Transformation to Intercept the Access to Instance Variables

MetaclassTalk delegates to the metaobjects the responsibility of controlling the access to the internal structure of the instances. As in the previous cases, this implies a shift from the base-level up to the meta-level. The technique we used to achieve this is also based on program transformation. Consider the following fragments:

The programmer's source code within a method, where a is an instance variable:

```
...
a + 10
...
```

is transformed into this:

```
...
self metaobject
        send: #+
        from: self
        to: (self metaobject atIV: #a of: self)
        arguments: (Array with: 10)
        superSend: false
        originClass: nil
```

Notice that acces to the value of a was transform into self metaobject atIV:#a of:self. In the same way, the programmer's source code within a method, where a is an instance variable:

```
a := 15
```

is transformed into this:

```
self metaobject atIV: #a of: self put: 15
```

3.4 Class Instance Variables In the MetaclassTalk Object Model

In standard Smalltalk, class instance variables are used to preserve the internal state of classes, in the same way that instance variables are used to encapsulate the state of instance objects. As classes are first class objects [Coi87, Coi90, Riv96], class instance variables are accessible only through the public interface provided by the classes; they are not available in the scope of their instances. There is also a second kind of variables available for classes, known as class variables, that can be shared within the class hierarchy.

So at the level of classes, there is the possibility to extend the structure of each class, separately of the definition of any other class—except for subclasses. This is a side-effect of the fact that in

Smalltalk, metaclasses are arranged in a hierarchy that replicates exactly the hierarchy of classes and thus each class is the singleton instance of its metaclass [Coi87, Coi90, Riv96].

However, in MetaclassTalk, several classes are likely to be instanciated from the same metaclass, for example StandardClass and thus, the same internal definition is available for all of them. Thus, adding a new variable to the definition of a metaclass, automatically introduces an extra variable in every instance (class) of the metaclass. Notice that this closes the gap between classes and instance object definitions, as in MetaclassTalk, classes are really treated as instance objects with an explicit class. That is the reason why we removed the #classVariableNames: keyword from the message to define new MetaclassTalk classes. In fact, intance variables, are supposed to be specified in the class of the object, which in this case, is a metaclass.

3.5 Summary

In this chapter described the implementation of MetaclassTalk in Squeak. We discussed the internal representation of objects and the mechanisms and techniques used to intercept method invocations. We also described how messages sent between objects are handled at the metalevel.

Part II

Be Mobile

Chapter 4 Mobile Computation

Nowadays, modern computer networks can no longer be considered just plain communication technologies that transport information. Rather, they constitute innovative media that supports new forms of cooperation and communication among users. The cause and effect of this important phenomenon has been the massive access to and availability of high-speed and low-cost networks on one hand, and the development of easy-to-use technologies on the other [FPV98, OHE96]. Clearly, this is changing the nature and role of networks, particularly the Internet, and has triggered the creation of new application domains and markets.

There have been many attemps to provide effective answers to this multifaceted problem. Most of the proposed approaches, however, try to adapt well-stablished models and technologies and usually are based on the traditional client/server architecture, like CORBA [OMG]. A different approach originates in the promising research area exploting the notion that *"a computation starting at some network node may continue its execution at some other network node"* [Car97]. *Mobile computation* not only involves *code mobility* but also *mobility of the internal state* of the computation; that is, data and execution state [Car97]. The following sections explore these concepts at some level of detail and the different technologies and approaches associated to them.

4.1 The Big Picture

According to work in the field of distributed systems and software engineering [OHE96, Whi96, FPV98], the next generation of software systems is likely to be built using *roaming agents* that come and go across the networks. This revolution promises to be just as traumatic as the one the industry went through when client-server applied a giant chainsaw to mainframe-based monolithic applications and broke them apart into client and server components [OHE96]. This section will tell us why, introducing and motivating the concept of mobile computations and roaming agents.

4.1.1 Current State

Today, our industry stands at a new threshold brought on by: *a)* the exponential increase of low-cost bandwidth on WANs, for example the Internet; *b)* a new generation of network-enabled, multithreaded desktop operating systems and *c)* and increasing availability of easy-to-use technologies accessible to naive users, for example the World Wide Web. All these together are changing the concepts of computer networks, that can no longer be considered just plain communication technologies used to transport information. Rather, they constitute innovative media that supports new forms of cooperation and communication among users [FPV98, OHE96]. Under this perspective, public networks are expected to provide a *platform* on which third-party developers can build new applications and forms of interaction [Whi96].

However, this evolution path is not free of obstacles and several challenging problems [FPV98] must be addressed before:

- Scalability: software systems and applications available for small networks are not applicable when scaled to highly dynamic and world-wide networks like the Internet.
- Customizability of services: different users have often different requirements. Systems should have the the ability to extend functionality without increasing size nor complexity of applications.
- Flexibility and extensibility: to afford the dynamic nature of both the underlying communication infrastructure and the market requirements.

Several approaches have been taken to provide solutions for the previous multifaceted problem. Most of them, trying to adapt well-stablished models and technologies based on the client-server architecture. A good example of this is CORBA [OMG] that integrates the wonders of object-orientation with a bunch of services and facilities to build distributed applications. The client-server architecture is based on the Remote Procedure Calling (RPC) model of interaction for computer-to-computer communication. Under RPC, an application running on one computer can invoke procedures in another application that runs on a remote computer. Each request or invocation might include some data as arguments for the procedure. The response, in turn, includes data which are the results of the invocation. Figure 4.1 depictes the situation.

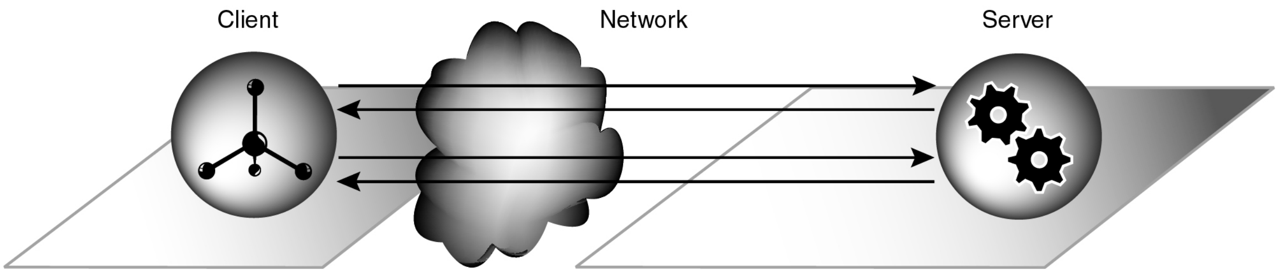


Figure 4.1 RPC's model of interaction

Unfortunately, client-server does not provide the flexibility, customizability and scalability required for WANs [FPV98]. One of the major disadvantages is that in RPC, and thus in client-server, *ongoing interaction requires ongoing communication* [Whi96]. Thus, a new paradigm is required, not only to cope with all this but also to open the doors to new kinds of interactions and functionalities.

4.1.2 The Next Revolution

As an alternative to client-server, a new and promising paradigm is emerging that exploits the notion of *mobility*, particularly, of *mobile computation*. This approach proposes to explote the mobility of code and execution state of software components across the network as a way to reduce remote interactions and provide new models of application development [Whi96, Car97]. The following definitions attemp to formalize and clarify the ideas.

Mobile computation refers to the notion that a computation starting at some network node may continue its execution at some other network node. Mobile computation involves much more than just moving code. *Code mobility* is useful but also limiting and some other elements are must be involved, such as *control mobility*, *data mobility* and *link mobility* [Car97].

Code mobility is the capability to dynamically change the bindings between code fragments and the location were they are executed [FPV98]. *Control mobility* occurs when a thread of control originating at some network node continues execution at some other network node. No code is moved in this process, just control [Car97]. A good example of control mobility is when a client sends a request to a server using RPC. In this case, the client is delegating the control on the server.

Data mobility concerns the movement of data over the network. This data must be on-line portable, that is, data structures are marshaled (converted to portable form) at the originating side, sent over the network, and unmarshaled at the receiving site into corresponding data structures, possibly within a different computer architecture [Car97]. *Link mobility* means that the endpoint of a network connection can be sent over another network connection. The receiving party is then connected to the other endpoint [Car97].

A *mobile system* is a computer-based system that supports some degree of computational mobility. Such a system should include at least a run-time environment and some policies to handle code and migration of the internal state of the computation. Unlike RPC, mobile computation involves the movement of code, not just the execution of code already available in remote network nodes. The other elements of mobile computation—control, data and link mobility—have already been widely studied and used; in fact all of them are already provided by RPC [Car97]. In mobile computation, control must move as well as code: the code that is transfered must be executed. Data must also move in order to preserve the state of mobile computations across moves. Finally, network links must also move, since they are part of the state of the computation (at least, in models of mobility that support remote connections) [Car97].

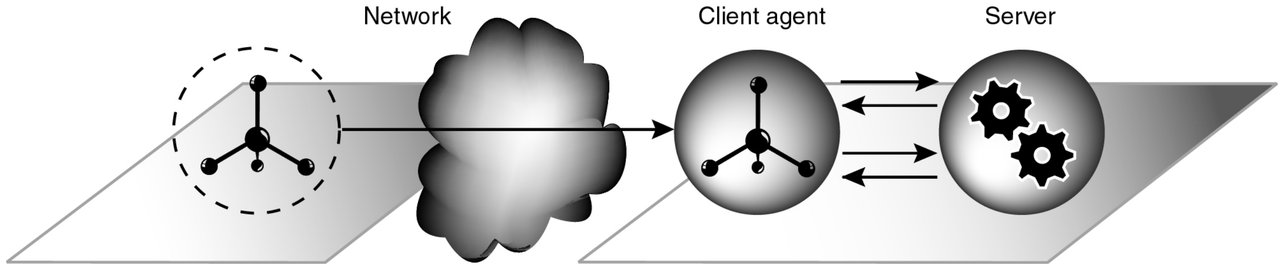


Figure 4.2. Mobile computation introduces a new model of interaction

As compared to RPC, this new paradigm views computer-to-computer communication as enabling one computer not only to call procedures in another, but also to supply the procedures to be performed. Importantly, these procedures can call procedures provided by the receiving computer. Note that such procedure calls are *local* rather than remote. Thus *ongoing interaction does not require ongoing communication* [Whi96]. Figure 4.2 depictes the situation.

4.1.3 Application Domains

The motivating force behind mobile computation are the expected benefits of it in a number of application domains of distributed systems [OHE96, Whi96, Car97, FPV98]. The purpose of this section is to provide an overview of the key benefits and domains of application which are being identified as suitable for the exploitation of mobile computation [OHE96, Whi97, FPV98, LO98]

4.1.3.1 Key Benefits

- Service customization: in client-server architectures, servers provide an *a-priori* fixed pool of services accesible through an *statically* defined interface which hardly meet clients needs of functionality. Every time an upgrade of functionality is needed, more complexity is attached to the server. An alternative approach would be the following: *a)* servers provide a bunch of low-level services that seldom need to be changed and *b)* clients create a component that is sent to execute locally on the server; the component integrates invocations to the server services and performs its own computation. This allows to extend funtionality from the client side. A special case of service customization is *protocol encapsulation*. In this case, when data are exchanged in a distributed system, each host owns the code that implements the protocols needed to encode outgoing data and interpret incoming data. The ability to create and send components that know how to code and manipulate such data, solves the problem of protocol evolution. At some extent, we can see this as an application of the Visitor design pattern [GHJV95].

- Support for software deployment and maintenance: some products, notably some Web browsers, use some form of program downloading to perform automatic upgrades over the Internet. This scheme allows the automatic distribution of new versions of software applications. In this case, only mobile code is involved.

- Increased performance by means of autonomous execution: RPC-based interaction implies a lot of network communication between clients and server to perform complex activities. The idea of packaging a conversation and dispatching it to a destination host, where the interaction can take place locally, is particularly suited for environments where network connections are slow or more expensive.

- Fault tolerance: remote interaction implies serious inconsistencies between the execution state of client and server in case of communication failures. Although there are mechanisms to recover from this, they are complex and involve more interaction. The fact that mobile computation reduces remote interactions, increases the tolerance to failures. In the worst case, only only local recovery is needed.

4.1.3.2 Domains of applications

At the time being, applications exploting mobile computation or at least code mobility can still be considered as relegated to a niche, at least compared to applications based on traditional client-server. This is a consequence of the inmaturity of the technology—mostly as far as performance and security are concerned [FTPV98]. The following is a review of some application domains expected to exploit in the near future [OHE96, FTPV98, Lan98].

- Distributed information retrieval: code mobility could improve efficiency by migrating the code that performs the search process close to the (possibly huge) information base to be analyzed. We prefer to move the computations to the data rather than the data to the computations.

- Active documents: mobile computation enables the embedding of code and state into documents and thus supports the execution of the dynamic contents when the documents are displayed.

- Advanced telecommunication services: services like videoconference, video on demand, or telmeeting, require a specialized middleware providing mechanisms for dynamic reconfiguration and user customization—benefits provided by mobile computation.

- Workflow management and cooperation (groupware): groupware supports the cooperative work of groups of persons and tools involved in an engineering or business process. Activities can be modeled as autonomous entities that circulate among the entities involved in a given workflow. Mobile computation provides support for this, as it encapsulates the know-how and the state.

- e-business: this domain refers to applications that enable users to perform business transactions through the network. A transaction may involve negotiation with remote entities and accessing information that is continuously evolving. Many researchers envision software components that embody the intentions of its creators, and act and negociate on their behalf [FTPV98].

4.2 Paradigms for Mobile Computation

The goal of software design is to develop software solutions. This is achieved by defining architectures that decompose software systems into smaller components and describe the interaction among them. Architectures that share similar characteristics are said to correspond to the same *paradigm*.

4.2.1 Basic Concepts

In the particular case of architectures for applications made out of mobile software components, the following concepts must be considered [FPV98]:

- Components: they are the constituent parts of a software architecture. There are different kinds of components: *a) Code components* that encapsulate the know-how of a computation; *b) Resource components* to model data or devices used during a computation and *c) Computational components* that carry out a computation

- Sites: represent the intuitive notion of locations where components can operate. Sites give the necessary support for the execution of computational components.

- Interactions: this abstraction is to model the messages exchanged as a result of inter-component cooperation and communication. Interactions among components residing in the same site are less expensive than interactions among components residing in different sites—due to the netwok communication involved in the latter case.

Notice that software architectures for mobile computation need to model explicitly the concept of location using the *site* abstraction to take into account the location of the components. As a side effect, interaction among components is likely to change dynamically as a consequence of a dynamic change of their location [FPV98].

4.2.2 The Paradigms

There are three design paradigms for software architecures that support mobile computation: *remote evaluation*, *code on demand* and *mobile agents*. The differences between them are determined by the technique used to move the code components, that is, the *know-how*, and the site where the computation actually takes place.

Following with an example provided in [FPV98], we base the explanations on the a metaphor where two friends—Louis and Christine—collaborate to make a chocolate cake. We decided to include the explanation of the client-server paradigm based on RPC following the same example—although it has already been discussed at the beginning—to compare the differences and help the understanding. Each paradigm is thus presented with an informal explanation based on the metaphor followed by a more formal description of the interaction among components. To link both explanations, Christine is represented with a component A living at a place S_A and Louis represented with another component B living in a place S_B. The recipe represents the know-how or program to provide a service; the cake represents the result of the service and the ingredients and the oven represent resources.

4.2.2.1 Client-Server using RPC

Louise would like to have a chocolate cake, but she doesn't know the recipe, and she does not have at home either the required ingredients or an oven. Fortunately, she knows that her friend Christine knows how to make a chocolate cake, and that she has a well supplied kitchen at her place. Since Christine is usually quite happy to prepare cakes on request, Louise phones her asking: "Can you make me a chocolate cake, please?". Christine makes the chocolate cake and delivers it back to Louise.

In a more formal description, in this case there is a computational component B (server) placed at site S_B advertising a set of services. Resource and code components, that is, resources and know-how needed for service execution, are hosted by site S_B as well. Another computational component, A (client), located at a site S_A requests the execution of a service from B (interaction). B performs the requested service by executing the corresponding know-how (recipe) and accessing the involved resources co-located with it (ingredients and oven). If a result is produced, it is delivered back to A with an additional interaction. Notice that the server has the know-how, resources and processor capability (Louise kows the recipe, provides the ingredients and oven and also makes the cake). Each request implies a remote interaction. So far, most distributed systems have been based on this paradigm. It is supported by a wide range of technologies such as RPC, Object Requests Brokers (CORBA) and Java's Remote Method Invocation (RMI) [Lan98].

4.2.2.2 Remote Evaluation (REV)

Louise wants to prepare a chocolate cake. She knows the recipe but she has at home neither the required ingredients nor the oven. Her friend Christine has both at her place, yet she doesn't know how to make a chocolate cake. Louise knows that Christine is happy to try new recipes, therefore she phones Christine asking: "Can you make me a chocolate cake? Here is the recipe: take three eggs…". Christine prepares the chocolate cake following Louise's recipe and delivers it back to her.

In REV, a computational component A has the know-how (recipe) necessary to perform the service but it lacks the resources required (ingredients and oven), which happen to be located at a remote site S_B, where another computational component B is located. Consequently, A sends the know-how to B, which in turn, executes the code using the resources available there. An aditional interaction is needed to deliver the result back to A (the cake). In this case, no resource components nor computational components are moved; just code. Java servlets are a good example of this paradigm, as they get uploaded in a remote Web browsers for execution.

4.2.2.3 Code on Demand (COD)

Louise wants to prepare a chocolate cake. She has at home both the required ingredients and an oven, but she lacks the proper recipe. However, Louise knows that her friend Christine has the right recipe and she has already lent it to many friends. So, Louise phones Christine asking: "Can you tell me your chocolate cake recipe?". Christine tells her the recipe and Louise prepares the chocolate cake at home.

In the COD paradigm, a component A is already able to access the resources it needs, which are co-located with it at S_A. However, A lacks the know-how (recipe) to manipulate such resources. The know-how is held by another component B living at S_B. A first interaction is needed for A to request the know-how to B and a second one to deliver the know-how from B to A (Christine tells Louise the recipe). In this paradigm, only code transfer is involved; no resource component nor computational component is moved. Java applets are a good example of this paradigm, as they get downloaded on demand and execute locally in the browser.

4.2.2.4 Mobile Agents (MA)

Louise wants to prepare a chocolate cake. She has the right recipe and ingredients, but she does not have an oven at home. However, she knows that her friend Christine has an oven at her place, and that she is very happy to lend it. So, Louise prepares the chocolate batter and then goes to Christine's home, where she bakes the cake.

In the MA paradigm, the know-how is owned by A, which is initially hosted by S_A. However, some of the required resources are located on S_B. Hence, A migrates to S_B carrying the know-how and possibly some intermediate results (chocolate batter). After it has moved to S_B, A completes the service (Louise bakes the cake) using the resources available there. The mobile agent paradigm is different from the other ones since the associated interactions involve the mobility of a computational component. In other words, in MA a whole computational component is moved to a remote site, along with its state, the code it needs, and some resources required to perform the task. Notice that in case, code components are not tied to a single host, rather, they are combined with some resource components and merged in a computational component that provides the processing capacibilites.

4.2.2.5 Some Discussion About the Paradigms

The most salient characteristic of the previous paradigms is that they model explicitly the concept of location. The *site* abstraction was introduced at the architectural level in order to take into account the location of the different components involved. This makes it possible to model the cost of the interaction. Particularly, interaction among components living in the same site is quite less expensive when compared with the interaction among components that require communication across a network [FPV98].

Another important advantage introduced here with respect to traditional approaches for software engineering, is that mobile computation paradigms provide flexibility to change dynamically the quality of the interaction—and thus reduce interaction costs. This is because mobile components are allowed to change their location through out their life cycle and thus, the quality of their interactions [FPV98].

Finally, while paradigms define the architecture of the mobile system and are to be considered during the design stage, technologies are involved during the implementation. It is important to notice that mobile technologies are somehow *orthogonal* with respect to the paradigms. However, some of them are better suited to implement applications designed according particular paradigms. As an example, consider the mobile agent paradigm implemented using strong mobility. In that case, the technology provides all the means to fully and easily exploit the paradigm [FPV98].

4.3 Technologies for Mobile Computation

For the purposes of this work, our definition of mobile computation involves both the move of code and the move of the internal state of the computation. It is important to remark that these concepts still generate some confusion in the field, and in fact, there is no general consensus on the semantics of mobile computation, terminology and related technologies. Certainly, confusion and disagreement are typical of a new and still immature research field [FTPV98]. This section brings some of the classifications and abstractions already made by Fuggetta, Picco and Vigna that allow an application to move code and state across the nodes of a network [FTPV98]. Mobile technologies include programming languages and their corresponding run-time support. They are used by application developers to implement distributed mobile software systems.

4.3.1 Constituent Elements

As opposed to conventional distributed system, that hide the underlying network providing a homogeneous and transparent layer of transport and interaction, in mobile computation the underlying network is not hidden; rather it is made manifiest, as shown in figure 4.3. In these architectures, the different elements involved are:

- Executing Units (EUs): active entities that represent sequential flows of computation. Some examples of EUs are the following: single-threaded processes, individual threads in a multi-threaded process, mobile agents.

- Computational Environments (CEs): layered on top of the network layer of each host, they provide an enviroment for the execution of the EUs. They have the capability to dynamically relocate their components on different hosts. A good example of this is the Java Virtual Machine [Sun].

- Resources: represent entities that can be shared among different EUs, such as a file in a file syetm, an object in a multi-threaded object-oriented language, network devices like printers, etc.

Resources can be *transferable* or *non-transferable*. A *transferable resource* can be migrated over the network. For example: a file. *Non-transferable resources*, however, cannot be migrated over the network. For example: a printer, a database. In turn, EUs can be decomposed in:

- Segment of executable code: provides the static description for the behavior of a computation. For example, a Java class.

- Data space: set of references to *external resources* accesible by the EU.

- Execution state or context: *private data* that cannot be shared, as well as *control information* such as the call stack and instruction pointer.

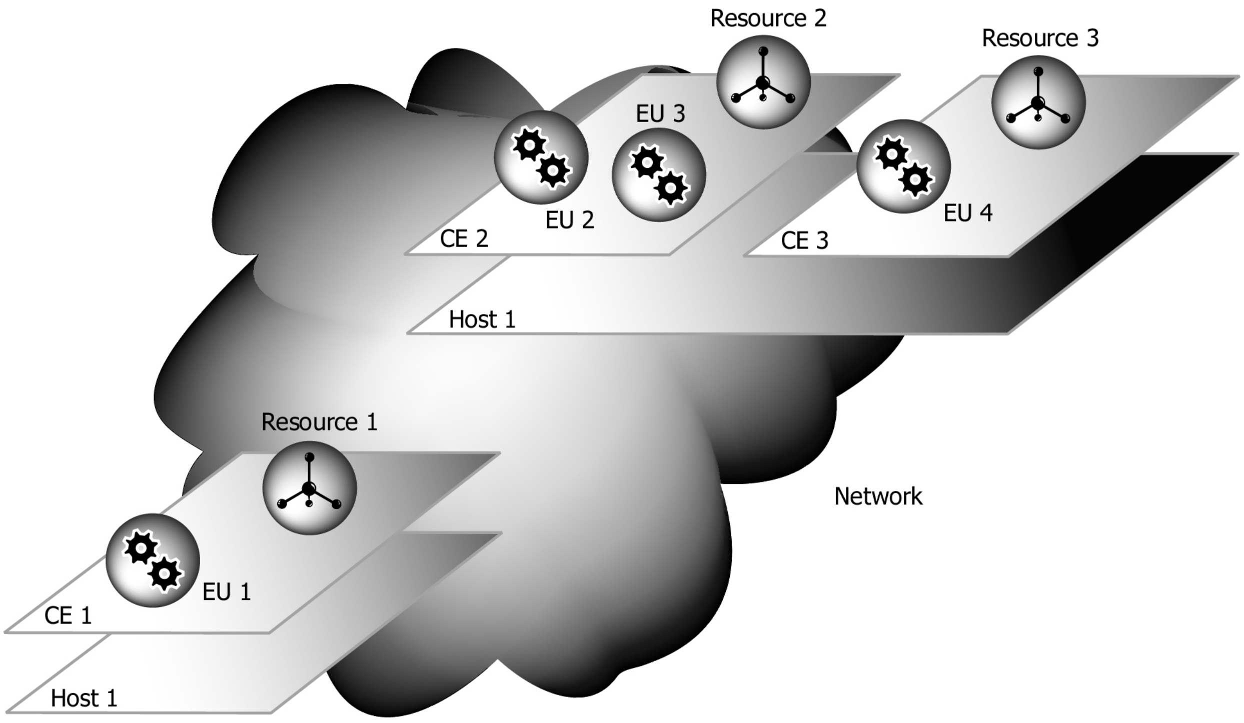


Figure 4.3. Technologies to support mobility explicitly represent the location concept, thus the programmer needs to specify where – i. e., in which CE – a computation has to take place.

4.3.2 Migration Policies

By *migration policies* we mean the mechanisms and techniques that perform the move of a given EU from the CE where it is currently executing to another one. In conventional systems, each EU is bound to a single CE its entire lifetime. In mobile computation, the code segment, execution state and data space of an EU can be relocated to a different CE. Based on the alternatives offered by existing systems, there is a wide range of alternatives to achieve this. For instance, several systems provide mechanisms that allow the programmer to pack some portion of the data space before the component's code is transfered. This is quite different from the situation where the system itself transfers the run-time image of the component as a whole, including its execution state (program counter, call stack, and so on) [FPV98].

There are two different aspects to consider: *a)* policies concerning the migration of code and execution state and *b)* policies concerning the administration of data and resources upon migration. Figure 4.4 shows a general classification for all this.

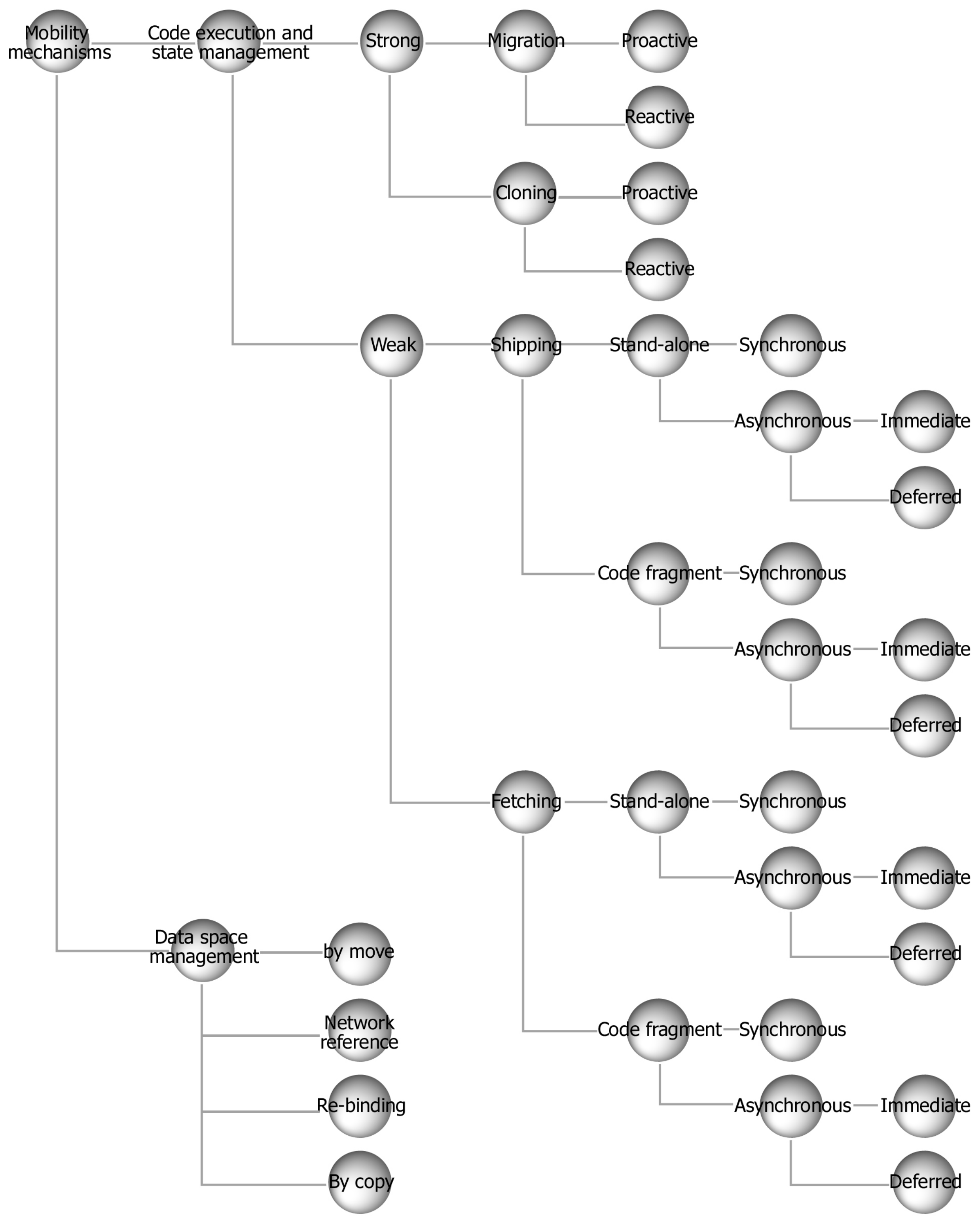


Figure 4.4. A classification of mobility mechanisms

4.3.2.1 Policies for Code and Execution State

According to what is migrated, we can make distinguish between *strong* and *weak mobility*. *Strong mobility* is the behavior exhibited by a mobile system that automatically handles the move run-time image of a given EU, to a different CE. The run-time image includes the code and the execution context. *Weak mobility* is the behavior exhibited by a mobile system that automatically handles the migration of code across different CEs; code may be accompained by some initialization data, but no migration of execution state is supported.
Actual implementations provide *strong mobility* by using *migration* or *remote cloning* mechanisms. Under migration, the executing EU is suspended, transmited to the destination CE and resumed. The EU at the original CE is removed. Mobility based on remote cloning, as suggested by its name, creates a copy of the EU at a remote CE without removing it from the original. On the other hand, depending on who—which entity—triggers the process, migration and cloning can be either *proactive* or *reactive*. Proactive means that the migrating EU determines by itself, autonomously, the time and destination for migration. Reactive, means that the move is triggered by another, external EU, different from the one that will be migrated.

Mechanisms supporting *weak mobility* provide the capability to transfer code across CEs and either link it dynamically to a running EU or use it as the code segment for a new EU. However, they do not provide any support to move the data and internal execution state. There are two mechanisms to transfer code: *fetching* and *shipping*. Fetching means that the receiving EU fetches the code to be dynamically linked and/or executed. In the case of shipping, EU at the origin sends the code to be executed to the destination CE.

When moving code, it becomes necessary to draw a distinction about the nature of the code being moved. In the case of a *stand-alone code transfer*, the code is self-contained and will be used to instantiate a new EU on the destination site. In the case of a *code fragment transfer*, the system sends just a fragment that is to be linked in the context of the already running code and eventually executed. This the case of Java applets, that allows to download bytecodes under demand.

Another important issue to consider refers to the synchronization of the transfer with respect to the execution of the EU that initiates it. With respect to this, mechanisms supporting weak mobility can be either *synchronous* or *asynchronous*. Synchronous means that the EU requesting the transfer suspends its own execution until the code is transfered and executed. Asynchronous means that the EU requesting the transfer continues with its normal execution during the transfer. In this case, asynchronous mechanisms, there are two policies to decide when the execution of the transfered code takes place: immediate and deferred. Under immediate execution, the code is executed as soon as it is received in the destination CE. Under deferred execution, the code is executed only when some predefinded conditions are satisfied—e.g., upon first invocation of a portion of the fragment or as a consequence of an application event.

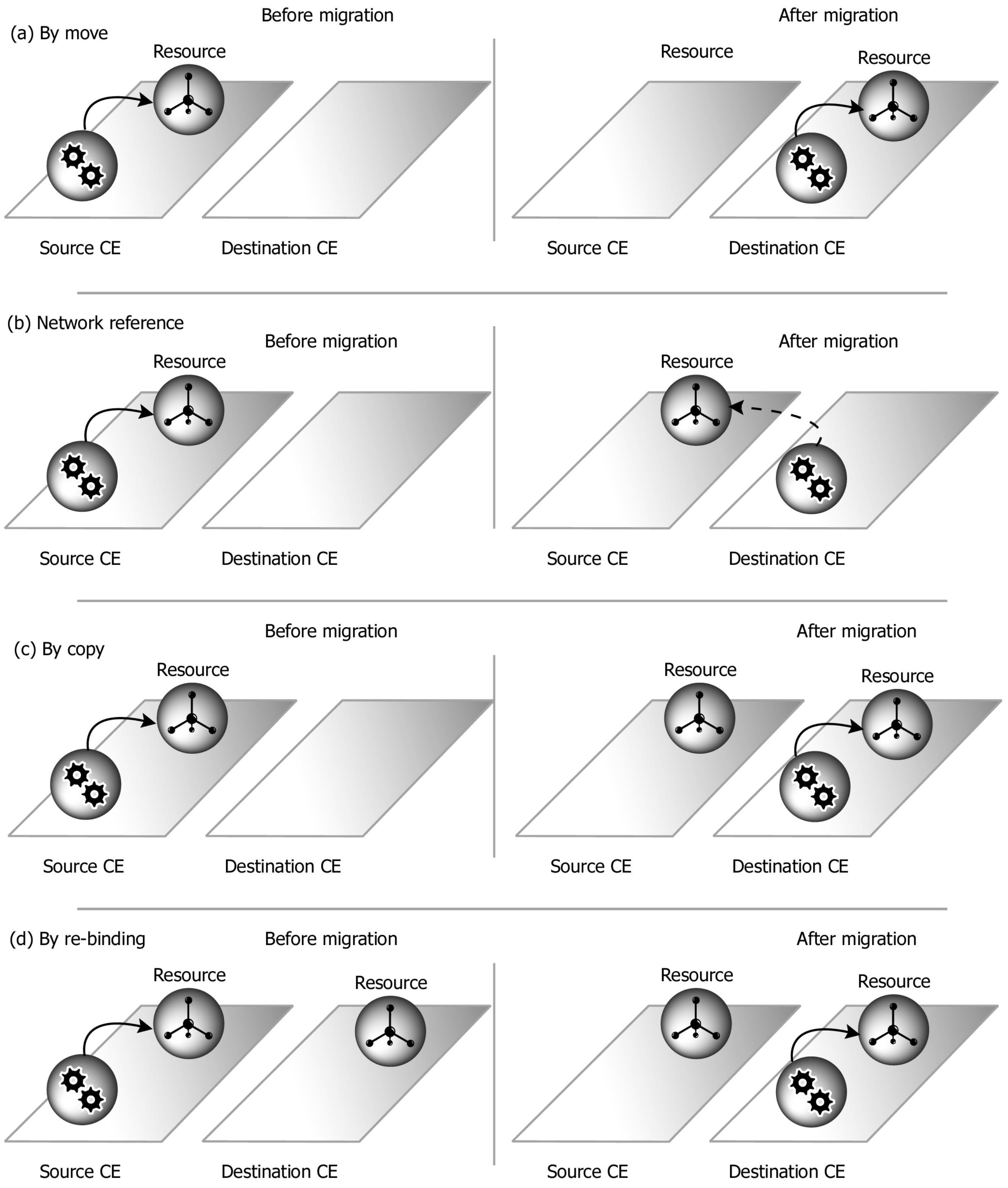


Figure 4.5 Mechanisms for data space management

4.3.2.2 Policies for Data Space Management

As stated before, every EU includes a data space, where it keeps references—called *bindings*—to external resources. When an EU has a binding to a resource, we say that the resource is *bound* to the EU. These bindings must be rearranged or restored upon migration, so that the EU can continue with its execution on the destination CE. There are several ways to achieve this: voiding bindings to resources, re-establishing new bindings or even migrating some resources to the destination along with the EU [FPV98]. According to the requirements of the application, it can be desirable to prevent the migration of *transferable resource*s. As an example, consider that for performance reasons, it might be undesirable to transfer a huge file or an entire database. For this reason, a transferable resource can be marked as *free* or *fixed*. In this way, while free transferable resources can be migrated over the network to another CE, fixed resources must be associated with the original CE. Figure 4.5 shows the different cases of data space management.

4.3.2.2.1 Types of bindings

Each resource can be described by the following attributes: *unique identifier*, *value* and *type* [FPV98]. These attributes are used to bound each resource to one or more EUs by these attributes. The following classification is provided:

- Binding by identifier: this is the strongest type of binding. The semantics of having a binding by identifier to a given resource is that at any moment, the binding must point at exactly the same resource, that is, the resource *uniquely* referenced by the identifier. As a consequence, the resource cannot be substitued. As an example, consider bindings to resources such as a databases—you want to access always the same pool of data.

- Binding by value: in this case, the reource associated to the binding could be eventually replaced with another one that has the same value. This means that at any moment the resource must be compliant with a given value, which cannot change as a consequence of migration. This is very useful when the EU is interested in the contents of a resource and wants to access them locally—we do not want to transfer the resource, but ensure its value. So, such resources should be available at any CE where the EU moves. An example of resources that may be bound by value is a calendar service.

- Binding by type: this is the weakest type of binding. In this case, the resource could eventually be replaced by another of the same type. The requirement is that at any moment, the bound resource is compliant with a given type, no matter the value or the identity. Network devices like printers are examples of resources that are usually bound by type. In that case, we just want to access the printer services, no matter its model nor technical details.

4.3.2.2.2 Relocation and Binding Reconfiguration

The above classification of bindings highlights two classes of problems that must be addressed by data space management mechanisms upon migration [FPV98]:

- Relocation of resources: this problem appears when resources must be migrated along with the EU. It concerns the techniques used to transfer and install resources at the destination CE.

- Binding reconfiguration: when an EU moves, the bindings it has to an external resource must be rearranged, independently if the resource is moved or not along with it. The reconfiguration involves two steps: *a)* unbound the resource from the original CE—for example graphical resources like images; and *b)* recreate the bindings at the destination.

Notice that the previous analysis considered only reconfiguration of the bindings held by an EU that is transfered to another CE. However, in a more general approach, we have to consider also the reconfiguration of bindings held by any EU when the associated resource is moved. In other words, we need to deal with *invalid* or *dead bindings*, that is, bindings that point at resources that are no longer available in the CE where they are supposed to be.

4.3.2.2.3 Policies for Data Space Management

There are several alternatives for data space management. Each of them implies different techniques to deal with the problems of the previous section. The way to tackle these problems is constrained both by the nature of the resources involved and the forms of binding to such resources. The following are alternatives to preserve the access to bound resources when the EU moves to another CE [FPV98]. Table 4.1 summarises the alternatives.

- Moving the resource: the resource is migrated along with the EU to the destination CE. The resource must be free transferable. In this case, we need to deal with both relocation and binding reconfiguration to reference the resource from the destination CE.

- Using a network reference: the resource stays at the original CE—no relocation—but bindings must be reconfigured using a network reference once the EU is transfered to its destination. Although this solution ensures that binding from other EU will stay consistent, it also implies additional traffic over the network everytime the transfered EU interacts with the resource. This often the only solution to deal with bindings by identifier, when the resource is not moved.

- Copying the resource: this very similar to moving the resource, with the sole difference that the resource is still available at the original CE. This is applicable for resources that can be replicated and the bindings do not need to worry about the identity of the resource.

- Re-binding the resource: when the resource is bound by value or by type and an equivalent resource is available at the destination, the best to do is recreate the binding with the resource available at the destination. No relocation is necessary and the interaction among the resource and the EU involves only local communication with the resource.

	Free transferable	Fixed transferable	Not transferable
By identifier	move network reference	network reference	network reference
By value	copy move network reference	copy network reference	network reference
By type	re-binding network reference copy move	re-binding network reference copy	re-binding network reference

Table 4.1. Binding, resources and data space management mechanisms

4.4 Enabling Mobile Agents with Objects

Previous sections presented a classification of the different technologies that give support for mobile computation. We saw that mobility can be provided at different levels (code, data, control, etc). On one hand, only mobility of code is allowed; on the other, the underlying system is able to effectively transfer the run-time image of any object. We also reviewed different paradigms available to define the architecture of software applications involving mobile computations. In this section we want to focus on the particular case of the *mobile agents paradigm* in the context of object-oriented programming.

4.4.1 Object Magic

Most people involved in the computer industry believe objects to be a *good thing*. An object is an encapsulated chunck of code that has a name, attributes and an interface that describes what it can do. Other programs can invoke the functions the interface describes or simply reuse the function itself. Objects should let us write programs faster by incorporating large chuncks of code from existing objects—this is called *inheritance*. In addition, an object typically manages a resource or its own data. We can can only access an object's resources using the interface the object publishes. This means objects *encapsulate* the resource and contain all the information they need to operate on it.

However, these classical objects only live within a single program. The outside world doesn't know anything about them and has no way to access them. They are literally buried in the bowels of a program. That is the reason why, some time later, objects were extended with capabilities to support interaction across the networks. This particular kind of objects are called *distributed objects*. Distributed object technology is the basis for remote interaction platforms like CORBA [OHE96].

Distributed Object = Object + Remote Interaction Capabilities

Although distributed objects provide a lot of advantages in terms of reuse and location transparency to develp distributed applications, their interaction, based on message passing, is similar to client-server based on remote invocation—we have already discussed the drawbacks of this model. An extension to this model is to provide objects with mobile capabilities. This idea of merging object and mobility is particularly appealing in domains like e-commerce [OHE96, FPV98, Lan98]. So far, we have already presented the mobile agent paradigm in a previous section, and based on that classification, a mobile object is similar to a *mobile agent* [OHE96, Whi96, Lan98, LO98, GB99].

4.4.2 Mobile Agents

For the purposes of this work, we are particularly interested in how object technology can be extended to deal with mobility. We call *mobile agent* an object or set of objects living in their own process or thread, that is able to temporally stop its execution and continue with it on a different location. The term mobile agent has been a bit overloaded as it has been used with different and somewhat overloapping semantics in both the distributed systems and artificial intelligence communities. We provide the following definitions extracted from several sources [Whi96, Lan98, LO98, GB99].

An *agent* is a computer program that acts autonomously on behalf of a person or organization. Each agent has its own thread of execution so tasks can be performed on its own initiative. A *mobile agent* is an object that can move between different execution environments; this means that a mobile agent is not bound to the system where it begins execution. It is meant to be completely self-contained and has the unique ability to transport itself from one place or location in network to another. Although it might communicate remotely with other agents, it can move to their location and communicate locally when it gets there.

When an agent transfers itself, the agent travels between execution environments called *places*. A place is a context within an agent system in which an agent can execute. This context can provide functions such as access control. The source place and the destination place can reside on the same agent system, or on different agent systems [Lan98, LO98, GB99].

An *agent system* is a platform that can create, interpret, execute, transfer and terminate agents. Currently, most agents are programmed in an interpreted language (for example, Tcl and Java) for portability. The ability to travel permits a mobile agent to move to a destination system that contains an object with the agent wants to interact. Moreover, the agent may utilize the object services of the destination system.

Mobile Agent = Distributed Object + Mobile Capabilities + Thread of execution

Notice that our definiton is different from Telescript mobile agents definition [Whi96]. In Telescript, agents are not allowed to stablish remote communication with other agents; and they just encapsulate behavior and data state but no references to external resources. Telescript's model probes to be more tolerant to network connection failures, as agents can survive autonomously without external interaction.

In previous sections, we took a more general approach to mobility and were dealing with executing units (EUs) that included a context private to each unit. In that contexts, each agent correspond to an executing units. We also saw that each unit includes some private data and some control information. The private data corresponds to the internal data or object state of an agent; the control information referes to the state of the thread where an agent executes (instruction counter, stack of frames and so on).

4.5 Requirements for Strong Mobility

The goal of this report is to analyze the different aspects that must be considered when defining a platform that provides support for strong mobility in an object oriented fashion. If an *object* is an entity that encapsulates both state and behavior, a *mobile agent* is an object or set of objects that can travel over a network.

4.5.1 Agent Management

This section provide platform operations for creating, migrating and killing agents. It should be possible to create an agent given a class name for the agent, suspend an agent's thread of execution, resume its threads or terminate it in standard way. The following is a more detailed coverage of these aspects.

4.5.1.1 Agent creation

An object gets created within an environment of execution (place). The creation can be initiated either by an object residing in the same place or by an object residing outside the place..

For each agent, there is a class from which the agent system instanciates an agent. The class definiton needed to instantiate the agent should be present on the environment where the object is to be created Agents are created in the place where the client application (source system) specifies. To create an agent, an agent system should:

- Start a thread for the agent
- Instatiate the class of the agent
- Generate (if necessary) and assign a globally unique name
- Start execution of the agent within its thread.

Every agent has an identifier that is unique during its lifetime (immutable) which assigned by the agent system. Our platform should give some kind of support so that it can be possible, given an identifier, to retrieve a reference to an agent. In the same way, given an agent, we should be able to extract its identifier.

4.5.1.2 Agent Disposal

How does a mobile agent ends its life? The disposal can be initiated by the agent itself, by another agent living in the same place, or by another agent living outside the local place.

4.5.1.3 Agent Migration

As stated previously, it is advantageous for two agents to communicate at the same place rather than across a network. So allowing a source agent to travel to a remote agent system achieves the benefit of locality.

When an agent transfers to another place, the agent system creates a travel request. As part of the travel request, the agent provides naming and adressing information that identifies the destination place. If the source agent system reaches the destination place (at a destination agent system), the destination system must fulfill the travel request, or return a failure indication to the agent. If the destination place cannot be reached, then a failure indication must be returned to the agent. When the destination system agrees to the transfer, the agent's state and if necessary, its code are transfered to the destination place. Then, the destination system reactivates the agent and execution is resumed. Migrating an agent includes the following actions: initiating the agent migration, receiving the at the destination place, transfering the necessary code to continue execution. Figure 4.6 shows the different stages.

4.5.1.3.1 Dispatching an Agent

When an agent is preparing to migrate, the agent must be able to identify its destination. With the destination available, the agent requests the source system for the transfer. As shown in figure 4.6 (left side) the different steps to transfer an agent to a remote system are:

- Suspend the agent: halt the agent's execution thread. Normally, a notification is sent to agent prior to its interruption

- Externalization is the process of translating a graph of objects into a stream of bytes which can be sent as a message over the network or written into a file on disk. The second step is thus to identify the pieces of the agent's state that will be transfered. In the case of strong mobility, the whole runtime image must be transfered. Once the state has been identified, the system proceeds to externalize it.

- Encode the externalized agent in a suitable format for the chosen transport protocol. For example: if the object is to be transfer over a TCP/IP connection, sent attached to an email, etc.

- Perform the transfer. For example: open the TCP/IP connection and send the data stream.

4.5.1.3.2 Receiving an Agent Transfer

Before an agent is received into a destination, the involved parties have already agreed the transfer, that is, the agent transfer was accepted at the destination. As shown in figure 4.6 (right side) the different steps to re-install the agent at its destination are:

The actions performed are:

- Receive the data

- Decode the agent: the data, received in a format dependent on the transport mechanism, is decoded and put in a suitable format that the system can manipulate (data stream).

- Internalize the agent from the stream. This consists in rebuilding the graph of objects

- Resume agent execution

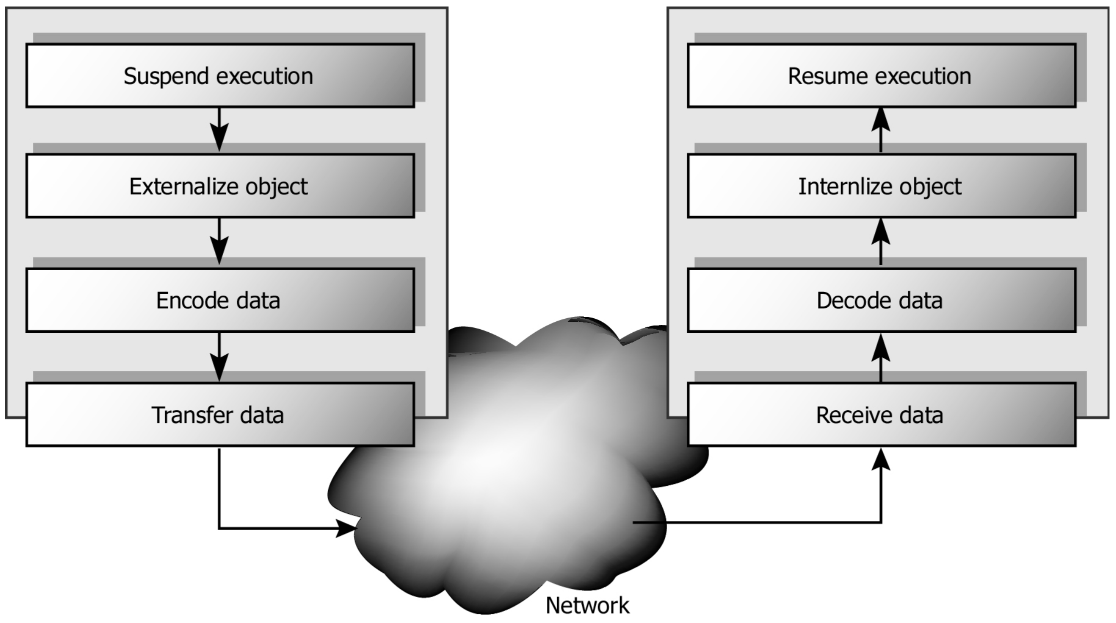


Figure 4.6. Transfering an agent

4.5.1.3.3 Class Transfer

Class transfer is the ability to transfer class information from one agent system to another. This ability is a requirement in agent systems that support object-oriented agents.

There three reasons why class transfer is needed during the life cycle of an agent:

- To rebuild the agent after the transfer: at the moment of internalization, the class of the agent is needed at the destination place to rebuild the agent. If the class does not exist at the destination, it must be transfered from the source system.

- To allow the agent create new objects: after an agent is instatiated, the agent often creates other objects. Obviously, the classes of these objects are needed for their instantiation. If any of these objects' classes are not available at the destination agent system, they must be transfered from the source system.

- Agent execution: classes provided the behavior of agents

When an agent system requests a class transfer, the agent system must be able to identify the class to another agent system. Another issue concerns the provider of the classes that are transfered. There are at least two alternatives. On one hand, the provider is the source agent system that initiates the agent transfer. On the other, classes are provided by centralized server that keeps a repository of classes. There are several approaches concerning class transfers:

- Automatic transfer of all possible classes: the source system sends all classes needed to execute the agent with each transfer request. This approach eliminates the need for the destination agent system to request more classes later. However, automatically sending all classes consumes more bandwith than necessary if any of the transferred classes are already available at the destination system.

- Automatic transfer of the agent's class, other classes transferred on demand: the source system sends the class of the agent to instantiate the agent with each transfer request. If more classes are needed after instatiating the agent, the destination system issues request the provider for these classes. This approach does not require the source system to determine all possible classes necessary before transfering an agent. It is also more efficient. However, transfers on deman might fail if the destination system cannot access the provider. This failure could happen, for axample, if the source agent system is a portable computer that has been disconnected since the agent transfer request was sent successfully.

- Transfer a list of names of all possible classes with the transfer request. in this case, the source system sends a list of classes names that includes all the classes necessary. Then, the destination system request the provider only the classes that are not available in its configuration.. This approach is efficient, but still requires the source system to know which classes the agent needs before making the transfer request.

4.5.1.4 Agent and Place Identification

In order to provide mobility and interaction among agents, we need to use names (identifiers) to uniquely identify places and agents. Because a mobile agent travels, an agent name must be unique across all the agent systems. Identifiers are used to update references, i.e. re-binding and to its remote resources or other agents. Agent systems may provide a naming service that assigns the unique identifiers. Related to naming service, is the possibility of finding an agent based on its name.

4.5.1.5 Message Sending

As stated previously, our approach to mobile agents supports references to remote agents. Thus we need to specify the characteristics of remote message sending. Every message sent to an object involves the following participants:

- The receiver: a mechanism is necessary to find the remote agent and send it a message. Location transparency techniques allow to treat references to remote agents in exactly the same way as references to local agents.

- Code of the method to be invoked: the class of the receiver provides the code that must be executed.

- Parameters: although the invocation of a method is rather simple, some problems arise when sending the parameters. There are two alternatives to pass the parameters: *by reference* or *by value*. Passing parameters by reference: this allows to share objects living in different environments. Passing parameters by value: a copy of the actual values is sent to the destination. Each agent system must define its own policies to specify how parameters are sent.

- Type of invocation: there are three different schemes of method invocation [LO98]: *a)* Now-type messaging: this the most popular and commonly used messaging scheme. A now-type message is synchronous and blocks further execution until the receiver of the message has completed the handling of the message and replies to it; *b)* Future-type messaging: a future-type message is asynchronous and does not block the current execution. The sender remains a handle, called *future*, which can be used to obtain the result. *c)* One-way-type messaging: this scheme is also asynchronous. The sender will not retain a handle for the message and the receiver will never reply to it.

4.5.1.6 Reference Management

An agent living in a given place can have references to some other agents living in the same or in a foreign place. According to the direction of the references and the role played by the agents involved, the following classification can be made. Figure 4.7 provides a graphical representation.

- Outgoing references: the references an agent has *to* some other agents. Outgoing references can be local if the referenced agent lives in the same place or can be remote if the referenced agent lives outside the boundaries of the place where the agent holding the reference lives.

- Incoming references: the references agent has *from* some other agents. As in the case of oitgoing references, incoming references can be local if the referent agent lives in the same place as the referenced or can be remote if the referent agent lives outside the boundaries of the place of the referenced

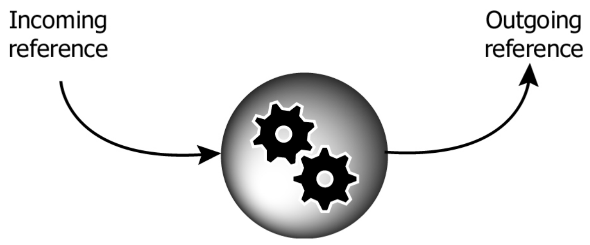


Figure 4.7. Incoming and outgoing references of an agent

In all cases, we need to define the mechanisms to deal with references on migration. Normally, outgoing references can be recreated in the destination, as an agent knows exactly the references it has. However, it is difficult to determine incoming references and then update them to point at the new location. We need to specify when and how this happens exactly. An important question that arises is if new references in the destination are to be created *before* or *after* deleting the ones in the source. In general terms we need to be able to:

- Create a remote reference from a local one: this is the case when an agent migrates and references to local agents become references to remote agents.
- Create a local reference from a remote one: this is the case when an agent migrates to the environment where some of the remote referenced agents live. Those remote references should become local references.

4.6 Summary

This chapter presented the concepts of *mobile computation* as a new paradigm to develop applications. At the end we provided the requirements to build a system that provides *strong mobility*.

Chapter 5 Reflective Facilities of Smalltalk to Implement Strong Mobility

Viewed from a high level of abstraction, Smalltalk is based on reified processes, and more generally on the objects needed to build a multiprocess system [Riv96]. The support for multiple and independent processes is provided with three classes named Process, ProcessorScheduler and Semaphore [GR83]. A Process represents a path of control in the system that runs independently of the paths of control represented by other Processes. A ProcessorScheduler schedules the use of the underlying virtual machine that actually carries out the actions represented by the Processes in the system. There are may be many Processes whose actions are ready to to executed and the ProcessScheduler determines which of these the virtual machine will carry out at any particular time. A Semaphore allows otherwise independent processes to synchronize their actions with each other. Semaphores provide a simple form of synchronous communication. The following sections explore in some detail the previous facilities offered by Smalltalk to enable strong mobility.

5.1 Processes

A process is a sequence of actions described by expressions and performed by the Smalltalk virtual machine. There are several processes running simultaneously in the system that manage time scheduling, event inputs such as keyboard/mouse, and regular user evaluations [GR83, Riv96].

A new process can be created by sending the message fork to a block. The actions that make up the new process are described by the block's expressions. As blocks may share an environment, independent processes use this facility to share objects. The message fork has the same effect on these expressions as does the message value. However, value returns only after the block has been completely evaluated, while fork returns immediately during block evaluation. In this way, the block evaluation and the expressions following the fork message execute independently.

In Smalltalk, each process is an instance of class Process. A block's response to fork is to create a new instance of Process and schedule the processor to execute the expressions it contains. Blocks also respond to the message newProcess by creating and answering a new instance of Process, but in this case, the interpreter is not scheduled to execute its expressions. This is useful because, unlike fork, it provides a reference to the Process itself.

During its lifecycle, a process may be in one of five states: *suspended, waiting, runnable, running* or *terminated* [HH95]. The first two states are almost similar: the difference between them is that a *suspended* process may be restarted to continue its execution, whereas a *waiting* Process cannot be restarted until it receives premission from a Semaphore. A Process is running when its actions are currently being executed by the interpreter. A process is *runnable* when it is scheduled for execution by the interpreter. Finally, a process is *terminated* when its execution can no longer be resumed.

The five messages that are of interest in forcing a Process to make a transition from one state to another are *suspend*, *terminate*, and *resume*, sent to a Process, and *wait* and *signal* sent to a Semaphore. The state transition diagram (figure 5.1) shows how these affect a Process. When *suspend* is sent to a Process, the process returns to the suspended state in which the processor is no longer executing its expressions. However, the actions represented by a suspended process can actually be carried out by sending the process the message resume. That message moves the process to the ready state. The third message, terminate, stops the advancement of a process forever.

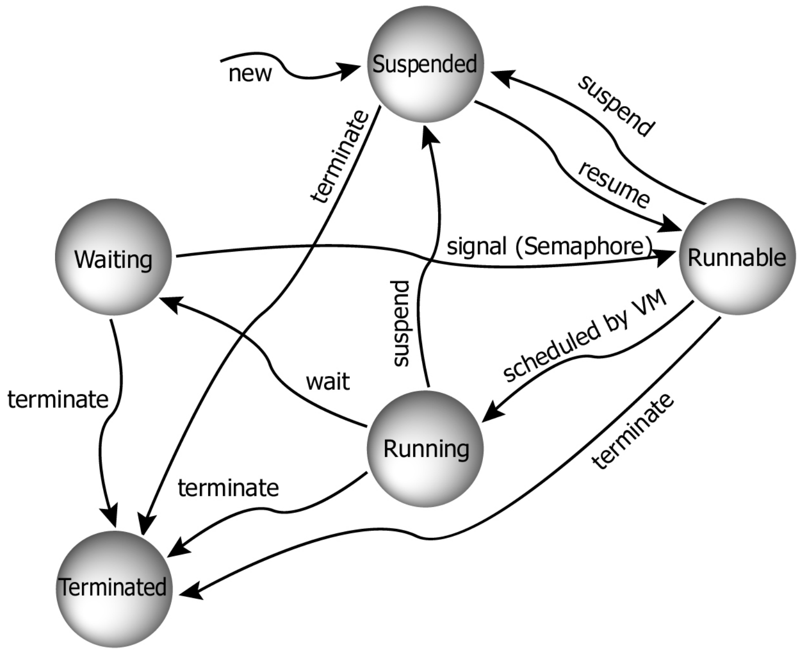


Figure 5.1 State transition diagram of Process

5.1.1 Scheduling Processes

The Smalltalk virtual machine has only one processor capable of carrying out the sequence of actions of the Processes. This means that there is actually only one running process at a time, which is identified as the *active* process. This processor, globally named Processor, is the sole instance of ProcessorScheduler class. Its mission is to coordinate the use of the physical processor by all Processes requiring service. Thus when a Process receives the message suspend or terminate, Processor selects a new active process among those that are runnable.

In order to provide more control for process scheduling, Processor uses a very simple mechanism based on priorities. A Process with a higher priority will gain the use of the physical processor before a Process with a lower priority. So the semantics between processes having different priorities are preemptive. Processes with the same priority as the currently active Process can have a chance to run when the yield message is sent to Processor. This message makes Processor suspends the active Process and places it on the end of the list of Processes waiting at its priority. If the list is empty, yield has no effect.

Finally, real-time scheduling is provided by the Delay class. It represents a real-time delay in the execution of a Process. The Process that executes a delay is suspended for an amount of (real) time represented by the resumption time of the delay.

5.2 Reification of the Model of Execution

The most remarkable reflective facility of Smalltalk is the reification of any process runtime stack through a chain of linked stack frames called contexts. Using contexts, a program is allowed to fully control its execution [Riv96]. A context is a snapshot of the internal state of the interpreter at a given time, in a way that the execution can be resumed later. In order to clearly understand how this works, we need to take a closer look at the underlying model of execution.

As discussed previously, Smalltalk applications are built as communicating objects that interact with each other by sending messages. However, internally such interactions are decomposed as sequences of eight-bit instructions, called *bytecodes*, that push and pop values to and from a stack of execution while control moves from one instruction to another. Two major entities are responsible to carry this out: a Compiler and an Intepreter. The Compiler receives the source code written by programmers and returns a sequence of bytecodes. The interpreter enables the execution of the bytecodes in the appropiate context. Thus it provides the semanctics of the interactions between objects.

5.2.1 The interpreter

The result of the compilation is a sequence of bytecodes that are stored in a CompiledMethod. According to [GR83], the interpreter can understand 256 bytecode instructions that fall into five categories: *push*, *store*, *send*, *return* and *jump*. In order to carry out the interpretation of the bytecodes, the following state information is kept within the interpreter:

- A CompiledMethod with the bytecodes that are being executed
- An instruction pointer to indicate the next bytecode to be executed in the CompiledMethod.
- The receiver and arguments of the message that invoked the CompiledMethod
- Any temporary variables needed by the CompiledMethod
- A stack

The execution of most bytecodes involves the interpreter's stack. *Push bytecodes* tell where to find objects to add to the stack. *Store bytecodes* tell where to put objects found on the stack. *Send bytecodes* remove the receiver and arguments of messages from the stack. *Return bytecodes* indicate the end of the current method evaluation and the value that must be returned. The return value is usually found on top of the stack, however there are four special return bytecodes to return *self*, *true*, *false* or *nil*. *Jump bytecodes* allow to alter the sequential execution of the bytecodes of the interpreter using conditional or unconditional jump instructions.

The interpretation of the bytecodes is carried out in a three-step cycle by the interpreter:

- Fetch the bytecode from the CompiledMethod indicated by the instruction pointer
- Increment the instruction pointer
- Perform the function specified by the bytecode

The remaining sections describe what might be called the data structures of the interpreter. Although they are objects, and therefore more than data structures, for the interpreter (VM) these objects are just data structure. The first two types of object corrspond to data structures found in the interpreters for most languages. *Methods* corresponds to programs, subroutines or procedures. *Contexts* correspond to stack frames or activation records. The final structure is that of *classes*, which is not used only by the compiler. Because of the nature of Smalltalk, the classes must be used by the interpreter at runtime to perform the method-lookup..

5.2.2 CompiledMethods

As we said before, the bytecodes executed by the interpreter are stored in instances of CompiledMethod. In addition to the bytecodes, a CompiledMethod contains some other parts: a *method header* and a *literal frame*. Figure 5.2 shows the structure of a CompiledMethod.

The method header is just a SmallInteger value that encodes certain information about the CompiledMethod, such as number of temporal variables, the size of the literal frame, number of arguments it takes and whether or not it has an associated primitive routine. The *literal frame* contains any objects that could not be referred to directly by bytecodes:

- Shared variables: global variables, class variables and pool dictionaries
- Most literal constants: numbers, characters, strings, arrays and symbols
- Most message selectors: those selectors that are not special, that is, are not encoded by the bytecodes themselves.

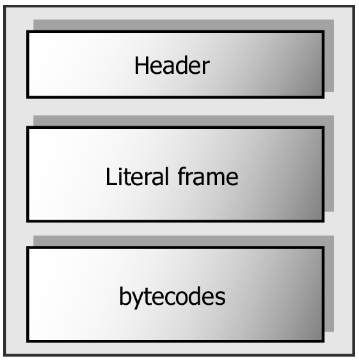


Figure 5.2 The structure of a CompiledMethod.

5.2.3 Contexts

Push, store and jump bytecodes require only small changes to the state of the interpreter. Objects may be moved to or from the stack, and the instruction pointer changed, but most of the state remains the same. Send and return bytecodes may require much larger changes to the interpreter's state. When a message is sent, all five parts of the interpreter's state may have to be changed in order to execute a different CompiledMethod in response to this new message. The interpreter's old state must be remembered because the bytecodes after the sending must be executed after the value of the message is returned.

The interpreter saves its state in objects called *contexts*. There will be many contexts in the system at any one time. The context that represents the current state of the interpreter is called the *active context*. When a send bytecode in the active context's CompiledMethod requires a new CompiledMethod to be executed, the active context becomes suspended and a new context is created and made active. The suspended context retains the state associated with the original

CompiledMethod until that context becomes active again. A context must remember the context that is suspended so that it can be resumed when result is returned. The suspended context is called the new context's *sender*.

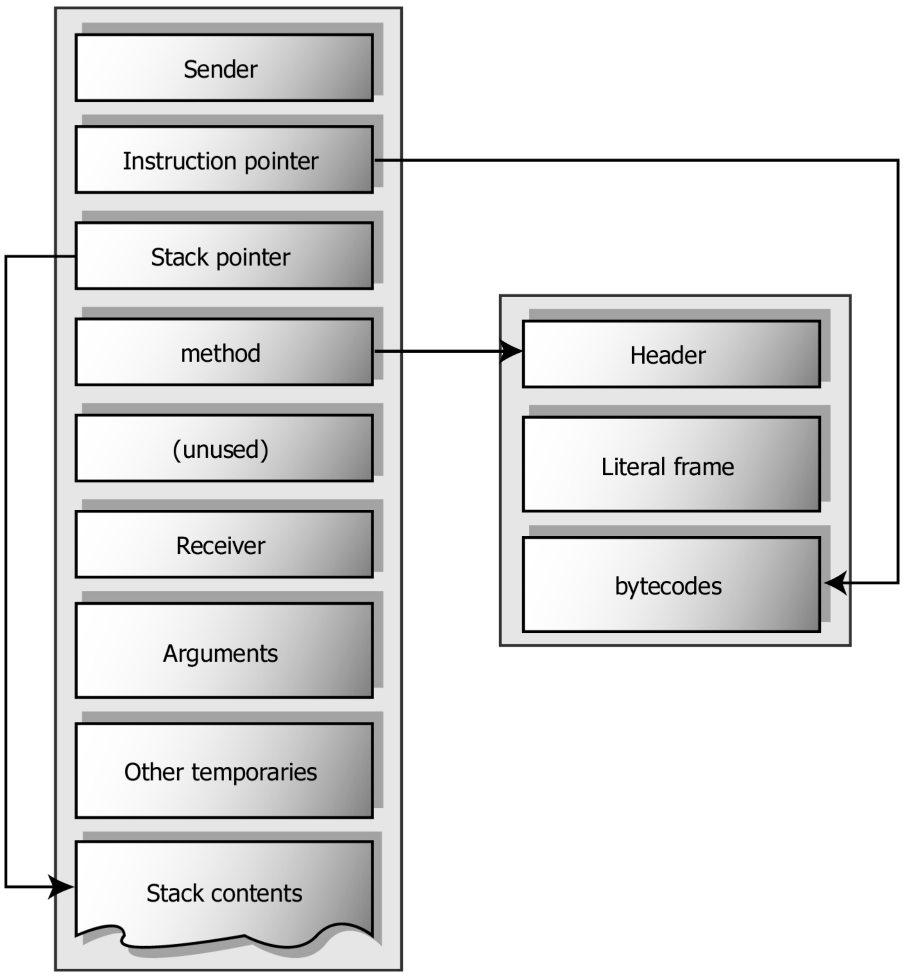


Figure 5.3. a MethodContext and its CompiledMethod

5.2.3.1 MethodContext and BlockContext

Context objects can be instance of any of the following classes: MethodContext or BlockContext. Instances of MethodContext represent the execution of a CompiledMethod in response to a message. Instances of BlockContext represent a block < […] > in a source method. BlockContext respond to the value: message to get evaluated with some arguments. When a BlockContext receives the value: message, it becomes the active context, which causes its bytecodes to be executed by the interpreter. Before this, however, the values of the arguments are pushed onto the BlockContext's execution stack. A BlockContext refers to the MethodContext whose CompiledMethod contained the block it represents. This is called the BlockContext's home. Figure 5.3 shows a MethodContext and its CompiledMethod. Figure 5.4 shows a BlockContext and its home.

The interpreter caches in its registers the contents of the parts of the active context it uses most often. These registers are:

- activeContext: this is the active context itself. It is either a MethodContext or a BlockContext
- homeContext: if the active context is a MethodContext, the home context is the same context. Otherwise, if the active context is a BlockContext, the home context is the contents of the home field of the active context. So this register will always point at a MethodContext
- method: the CompiledMethod that contains the bytecodes the interpreter is executing.
- receiver: this is the object that received the message that invoked the home context's method.
- instructionPointer: this is the byte index of the next bytecode of the method to be executed
- stackPointer: this is the index of the active context containing the top of the stack.

Whenever the active context changes (when a new CompiledMethod is invoked, when a CompiledMethod returns or when a process switch occurs), all of these registers must be updated.

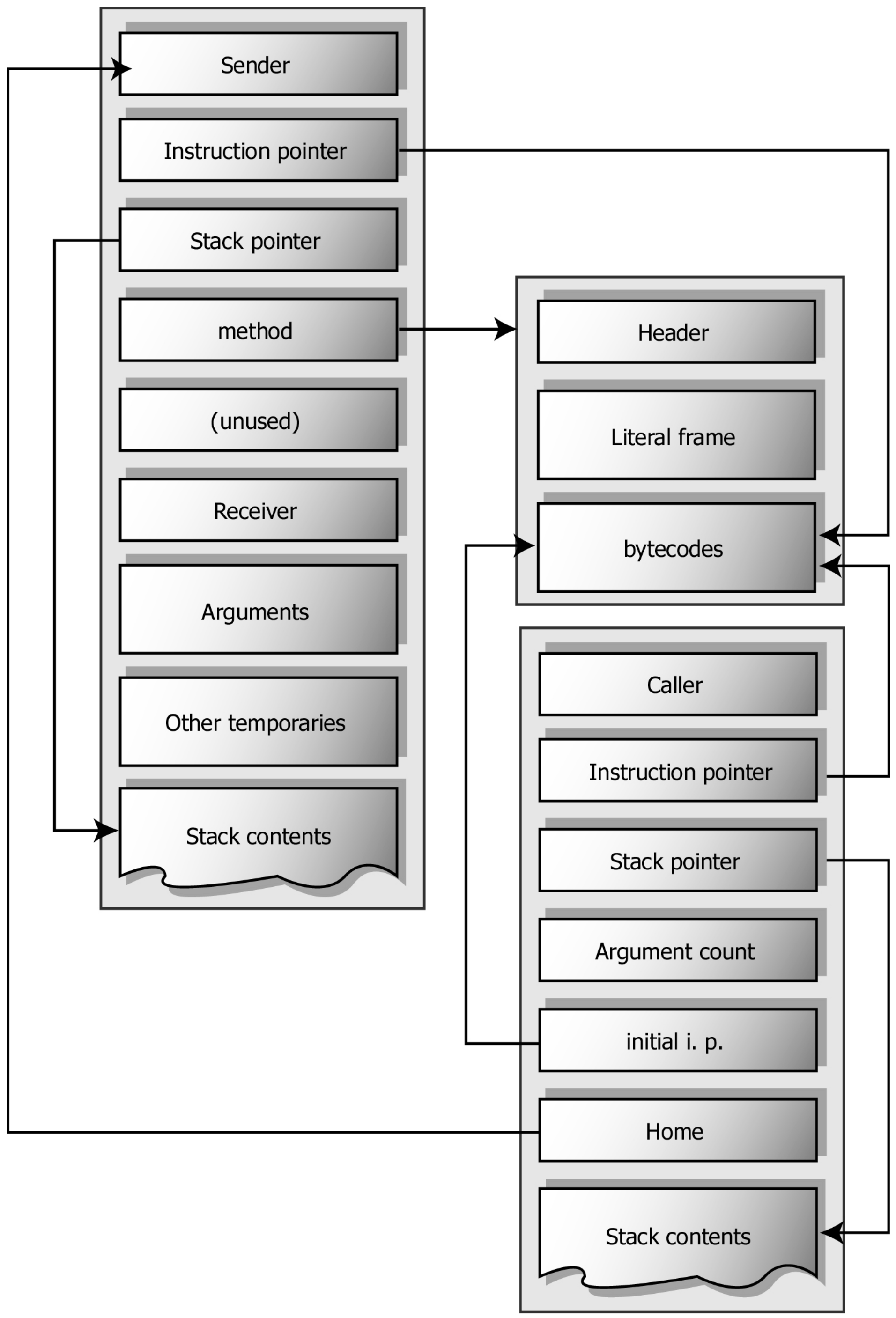


Figure 5.3. a BlockContext and its home

5.2.4 Classes

The interpreter finds the appropiate CompiledMethod to execute in response to a message by searching a method dictionary. The method dictionary is found in the class of the receiver or one of the superclasses of that class. In addition to the method dictionary and superclass, the interpreter uses the class's instance specification (fotmat value) to determine its instances' memory requirements. The interpreter includes the following registers related to classes

- messageSelector: this is the selector of the message being sent. It is always a Symbol.
- argumentCount: this is the number of arguments in the message currently being sent. It indicates where the message receiver can be found on the stack since it is below the arguments.
- newMethod: this is the method associated with the messageSelector.
- primitiveIndex: this is the index of a primitive routine associated with the newMethod if one exists.

5.3 Object Externalization and Internalization

Serialization is a process by which an object and the objects reachable from it (object graph), are encoded and stored to a stream of bytes, in such a way that the object graph can be reconstructed later from the stream. The act of serializing an object to a stream is called *externalization*. The act of reconstructing an object from a stream is called *internalization*. A *stream* is a data holding area with an associated cursor. A *cursor* is a mobile pointer that moves forward and backward as data is written and read to and from a stream. The data holding area can be in memory, on a disk file or across a network—we can't tell the difference. We *externalize* an object to a stream to transport it to a different process, machine, ORB, etc. We *internalize* an object when we need to bring it back to life at its new destination.

Serialization is normally used for lightweight persistence and for communication via sockets or Remote Method Invocation (RMI) in Java. Our goal in this section is to explore some of the facilities available in Squeak to use serialization for object migration.

5.3.1 Serialization in Squeak

Squeak offers serialization services through a group of classes named DataStream, ReferenceStream, SmartRefStream and DiskProxy. The former three extend the functionality of Stream class offering save-to-disk facilities to any object. The latter is an externalized form of an object to write on a DataStream.

DataStream provides the basic serialization mechanisms but cannot deal with the serialization of graphs of objects that include cycles. ReferenceStream is a subclass of DataStream that can effectively store one or more objects in a persistent form (disk file), including shared objects and cyles. However, ordinary ReferenceStreams assume that the layout of instance variables in an object on the disk is the same as the layout of that class in memory at the moment of externalization. They also assume that the class has the same name as before. SmartRefStream address these issues allowing to internalize objects whose instance variables or class name have changed. In order to achieve this, SmartRefStream records the names of the instance variables of all outgoing classes and makes the necessary adjustments when the file is read in.

The procedure to externalize an object is quite simple. All that needs to be provided is a name for the output file and the object to serialize. Consider the following example:

```
stream := ReferenceStream fileNamed: 'myObject.obj'.
stream nextPut: myObject.
stream close.
```

To get the object back to life is also simple:

```
stream := ReferenceStream fileNamed: 'filename.obj'.
myObject := stream next.
stream close.
```

Although the previous mechanism ensures that after internalization, the internal state of the object is effectively restored, the result is in fact a clone of the original object. Very often, this side-effect is not desirable if, for example, the same object is internalized several times during the same sesion. In that case, internalization should result in a reference to the original object already available in memory. In some cases, it is even difficult, impossible or not desirable to serialize the object simply by snapshotting and later reloading its instance variables (like a CompiledMethod or a Picture). In those cases, instead of externalizing an object itself, we need to externalize a reference to an object so that internalization results in the reconstructed reference.

The solution to this is provided by DiskProxy. The idea is to define a class method, named c*onstructor*, for each kind of objects that need special externalization (like CompiledMethods). Then, we build a DiskProxy that externalizes a message that performs an invocation of the constructor. Finally, during internalization, the message gets evaluated, resulting in the invocation of the constructor which in turn, returns the object. As an example, consider the following DiskProxy that externalizes the constructor to retrieve a CompiledMethod.

```
className := aCompiledMethod who first asSymbol.
methodSelector := aCompiledMethod selector.
DiskProxy global: #CompiledMethod
        selector: #atClass:selector:
           args: (Array with: className with: methodSelector)

CompiledMethod >>
atClass: aClassName selector: aSelector

        ^(Smalltalk at: aClassName) methodDict at: aSelector ifAbsent: [nil ]
```

5.3.2 External Interoperation

An important issue to consider is the interoperation between objects living in different address spaces. Currently, Squeak does not provide a framework to develop distributed applications, built out of distributed objects. However, it does includes all the necessary infrastructure to support network connections and communication.

The main abstraction available is the Socket class, which represents a network connection point. Current sockets are designed to support the TCP/IP and UDP protocols, although UDP is not yet implemented. Subclasses of socket provide support for network protocols such as POP, NNTP, HTTP, and FTP. Sockets also allow to implement custom services and may be used to support Remote Procedure Call or Remote Method Invocation in the near future. So we came to the conclusion that—with some work—it should be possible to design implement a framework on top of those basic features, to introduce more sophisticated mechanisms of interoperation like proxies.

5.3.3 Summarizing

The previous sections described the reification of processes and serialization mechanisms found in Smalltalk. They are the basis on top of which we can implement strong mobility. This section moves on that direction and attemps to formalize the results of some experiments. The goal if to explore the feasibility of implementating strong mobility with the the facilities available.

In general terms, the migration of an agent to a remote location is a three-step procedure. First, we encapsulate the object's complete state. This involves the encapsulation of the internal state described by the instance variables and the encapsulation of the execution state, that is, the Process where the object is executing. Next, we transfer this capsule to its destination using some transport mechanism. Finally, we bring the object back to life by restoring its state at the destination and reactivating its execution.

5.3.3.1 Memento Mori—Prepare To Die

The first step toward strong migration is to freeze and pack both the execution and the internal state of the migratory object. Our assumption that migratory objects run on their own threads allows to freeze them at any given time, as Smalltalk reifies their execution into Processes.

Two parameters are required to perform any migration: *a)* a reference to the destination place and *b)* a reference to the migratory process. With those parameters, the external object, called *migrator*, that actually performs the migration first freezes the execution of the migratory process. One way to do this is by sending the migratory process the *suspend* message. An alternative is to allow the migrator to run at a higher priority; this would ensure that the migratory process is effectively frozen—because of the preemptive scheduling model of Smalltalk. We encourage the use of the former alternative. Notice that our assumption that objects run at most in one thread at a time, ensures that their internal state is not affected by interaction with other objects.

The second action the migrator performs is the externalization of the process to a stream. Although externalization is no longer a problem and we have already explored the facilities available in Squeak, we need to make an inventory of the features that need to be externalized. Usually, Contexts include references to global objects like classes, Smalltalk, Processor, etc. Those objects should not be externalized. Rather, we need to transfer references to them that should be restored at the destination. So at some extent, we need a specialized externalization service that automatically packages certain objects as references. However, those serialization policies should concern only mobility; we do not want to affect the migration of the other applications using serialization.

More specifically, we need to specilize the method #objectForDataStream: but only during serializations started by the migrator. The idea is to define a class MobileReferenceStream, subclass of SmartRefStream, that overrides the method #nextPut: so that it sends the message #objectForMigrationDataStream. The default implementation of #objectForMigrationDataStream in Object returns the receiver, while in the case of ProcessScheduler and SystemDictionary the method should return an instance of DiskProxy encoding the callback to rebuild reference.

After externalization, the process can be safely killed at its source place. Killing a process is acomplished by sending it the terminate message.

5.3.3.2 Abduction—The Trip to Another World

The next step after externalization, is to send the process encapsulated in the data stream to the destination place. The sending of the stream to the destination is achieved by using some of the available mechanisms based on network communications.

At the time being, we suppose that all the classes referenced from the graph of objects encapsulated in the stream are available at the destination, with the appropiate versions of methods and internal definition. However, in Smalltalk classes may be also serialized and sent to the destination to get installed there. In this case, we need something else than a *file-in* mechanism that exports only the static definiton of a class. Classes are real objects and thus they also have an internal state. In next chapter, we explore an alternative approach for all this that address the issue of code mobility.

5.3.3.3 Resurrection—Back to Life

At the destination, the Process gets reconstructed from the stream. The internalization of the DiskProxies automatically restores the references to global objects. To resume the execution, the Process is sent the resume message. That brings the object inside the Process back to life, exactly at the point where it was frozen.

5.4 Summary

In this chapter we explored the the reflective features available in Smalltalk to implement a system that provides strong mobility. We claim that Smalltalk reifies as objects all the necessary entities that need to be manipulated.

Chapter 6 Towards a Model for Strong Mobility

This chapter discusses some design issues and choices towards the definition of a model that allows to develop applications made out of agents in Smalltalk. Our analysis is based on an abstract model that was designed to extend the Java language with facilities for strong mobility [BDLS00]. Although the resulting model is, of course, not exhaustive, and at some extent could be considered as an analysis, it shows the direction to follow to build a complete infrastructure for strong mobility. We identify two major components: *clusters* and *proxies*. Clusters provide all the functionality of agents and represent the units of migration. Proxies, on the other hand, act like local representatives of remote clusters. We also provide some ideas and examples about the way in which inter-cluster communication should take place. The second part of the chapter provides some guidelines for an implementation using MetaclassTalk.

6.1 Design Choices

6.1.1 Clusters

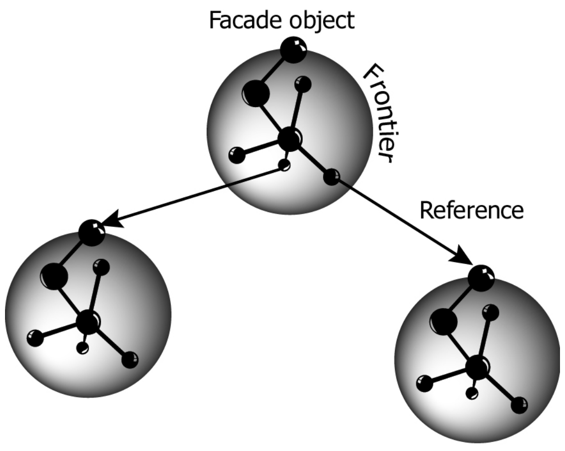


Figure 6.1 Clusters include a frontier, a facade and a set of objects within the frontier. A cluster can include references to the facade of another cluster.

Our design introduces the concept of cluster [BDLS00, HH95] as the minimal unit of migration. A cluster is similar to an agent is the sense that it can migrate from one location to another. However, a cluster is not a single object but rather a group of objects that constitutes a single unit of migration. This means that we deal with entities constituted by objects that are encapsulated as a whole and migrated atomically to the same destination. Figure 6.1 shows the internal structure of clusters. Each cluster is characterized by the following attributes and properties:

- a name that is a unique and immutable identifier assigned to the cluster at creation time. This name allows to identify the cluster through out the whole system.

- a place where the cluster is currently living. The place provides the resource to lodge all constituent objects of the cluster.

- a unique facade that acts as a distinguished object. All external interaction with the cluster goes through its facade, which constitutes an entry point for the services offered by the cluster.

- a frontier which is a border that indicates which objects belong to the cluster. The frontier of a cluster determines exactly the set of objects that are accesible from the facade of the cluster. The frontier defines the set of object that are migrated with the facade. Our idea is that some flexibility should be available to let the frontier change dynamically according to the needs of a cluster.

Clusters are defined by application developers who choose the classes of the objects that play the role of facades and the corresponding set of methods that constitute the interface of the cluster—the set of method that are accesible from the outside. The frontier provides the cluster's encapsulation. This means that other cluster cannot keep a references pointing at objects inside the frontier of other clusters other than the facade—as shown in figure 6.1. However, within the limits of the frontier, constituent objects can be mutually referenced and interact without any limits, including the facade. On the other hand, it should not be possible to build clusters out of other clusters (composition). As a consequence, all clusters are located at the same level and are globally visible, independently of their location. Finally, we take the concept of clusters is valid to model external resources like files, printers and other network devices. As each cluster provides its own migration policy, that varies according to the real possibilities of the resource they represent, we provide a uniform and homogeneous view. In the case of a printer, no migration is available; however, the encapsulation of it within a cluster brings homogeneity to the model as its services are made available to other clusters.

6.1.2 Proxies

As cluster can have references to other clusters, we need a mechanism to deal with that requirement. Our choice was to use proxies to model that. Proxies [GHJV95] are used as local representatives for clusters living in a different address space. Proxies provide some abstraction of the underlying communication mechanisms, enabling a transparent remote interaction. In fact, clusters always interact locally, either with other clusters living in the same place or with proxies that represent remote clusters. Proxies are created implicitly by the system everytime a reference to a new remote cluster is needed. Each proxy points at only one remote cluster. The same proxy is reused to handle the communication needs of several clusters living in the same place that interact with the same remote cluster.

6.1.3 Inter-cluster Communication

Smalltalk objects interact under a synchronous model of communication, allowing the same object to be involved at the same time in several threads of execution if it is attending requests from objects running in different threads. Although this model introduces a lot of flexibility and simplicity to develop concurrent applications with objects, it makes very difficult to handle migration of entities that are involved in several threads of execution. On the other hand, agents, as oppossed to objects, are pro-active and autonomous entities that run on their own thread.

The considerations above, influenced our decision about the model of interaction and execution of clusters. We borrow the approach from [BDLS00] which requires each cluster to run within its own thread of execution. In order to ensure that each cluster is involved at most in one thread at the same time, the interaction model for inter-cluster communication is asynchronous. However, inside the frontier, objects interact with each other using a synchronous messaging scheme, which is the same already available in Smalltalk.

More specifically, the interaction between two clusters is divided in two stages:

- Sending: the cluster playing the client sends a *request message* to another cluster playing the server. The request includes all the parameters and also some information to locate the client. As soon as the request is dispatched, the client's execution is released. However, instead of continuing with its normal execution, the client keeps on waiting for the answer in its own thread.

- Replying: at some moment, the pending request is served by the destination cluster. After the execution of the service, the cluster sends a *reply message* back to the source with the result—all the information to locate the sender was included in the request. Immediately, after the message is sent, the cluster continues its normal execution. When the reply arrives at the source place, the client processes the message and continues with its normal execution.

Notice that although the interaction occurs in an asynchronous fashion, the client is blocked in its own thread until a reply is available—the client continues its normal execution only after the reply arrives. So we can consider that this is a *pseudo synchronous* interaction. The main advantage of this approach is that it allows both parties to execute independently in their own threads, even when they interact with other clusters. Thus we have self-contained entities that encapsulate their object state and run autonomously.

This decoupling between sender and receiver is also an advantage to deal with migration [BDLS00]. It allows the cluster to receive requests while it is at one place, then migrate, and after send back the answers from it new location. This means that all requests received before migration are migrated along with the cluster. However, all incoming requests, arriving while the cluster is moving, are temporaly suspended by the source place and re-sent after, when the cluster is installed at the new location. An alternative for this is to send back a migration notification to the clients with the new location of the cluster. Figure 6.2 shows the configurations before and after a cluster C1 living in a place X migrates to a place Y. The scenario presents the situation where C1 receives request while it is in X and send the replies from Y.

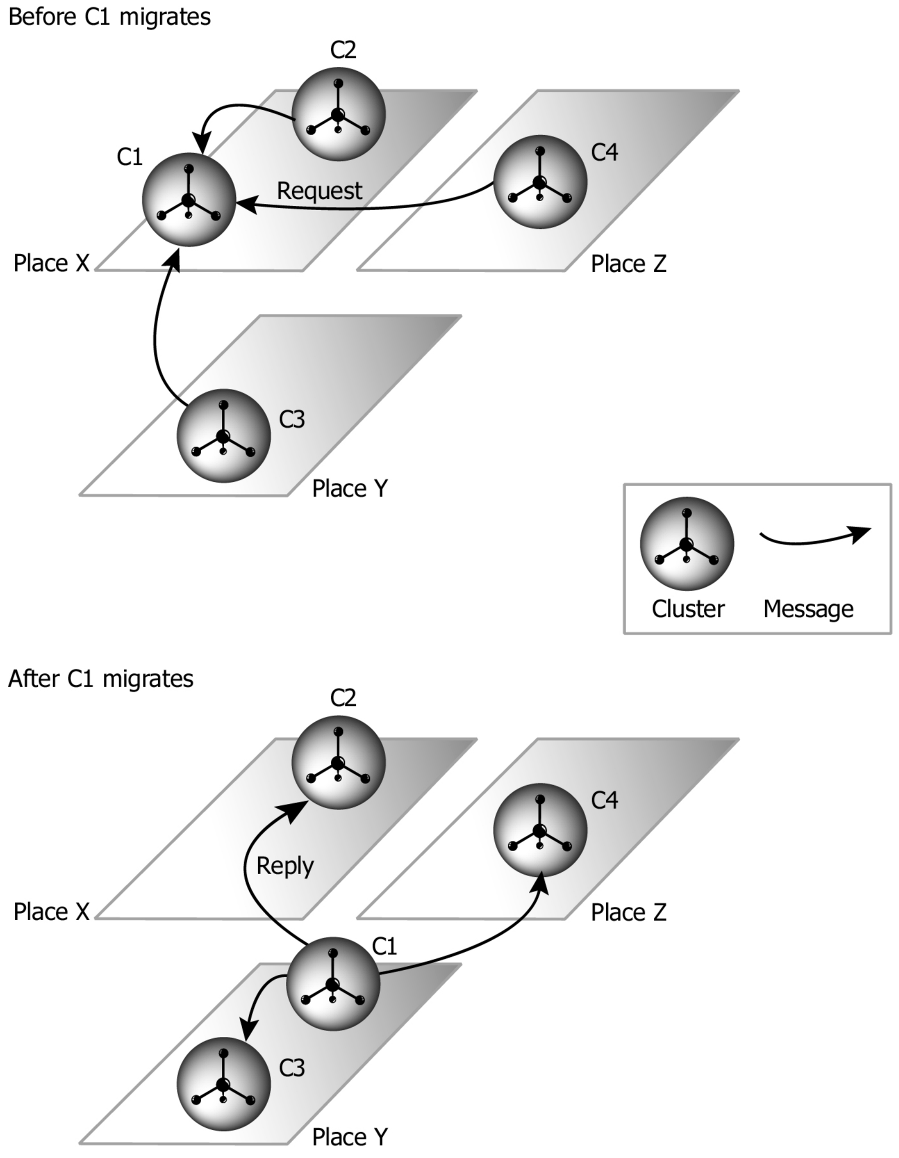


Figure 6.2. Inter-cluster communication allows a cluster to migrate and send replies from another location—different from the one where the request where received

6.1.4 Binding Reconfiguration after Migration

Once a cluster is migrated, before it can be re-started the references to other remote clusters must be updated. This problem, known as *binding reconfiguration*, was discussed earlier. In this model, binding reconfiguration is based on the use of proxies. We need to consider the problem in two different directions. On one hand, we need to deal with the outgoing references of the migrating cluster. On the other, we need to take care of the incoming references that automatically become inconsistent after migration.

6.1.4.1 Outgoing References

When the cluster is reinstalled at the destination place, old outgoing references to clusters that are now available in the same place are converted into local references. Old outgoing references to clusters that are now remote, require a proxy. This update of references involves interaction with the local place, as it is the entity that can return the references to other clusters and create new proxies as well.

It is important to remember that each place has at most only one proxy to represent each remote cluster. In other terms, there is no possibility to have two proxies in the same place pointing at the same remote cluster. This approach prevents problems of consistency upon migration if several copies of the same proxy are available, and at the same time, it minimizes the use of system resources.

Figure 6.3 shows the configuration of the references before and after migration of cluster C1 from a place X to another place Y. Notice how proxy reference proxyX4 is converted into a direct reference at the destination place. In the opposite case, direct reference to C3 requires a proxy in the after migration. Finally, proxy PX7 is replaced by PY7 at the destination.

6.1.4.2 Incoming References

When th cluster is reinstalled at the destination place, incoming references must be carefully updated whenever possible. We can easily detect incoming refereces at the source place and at the destination place, as the checking of outgoing references of clusters living there is not expensive. However, checking references coming through out the system is a quite expensive operation. In this case, we adopt a policy of notification upon demand, that is, the next time the owner of the reference attemps to access the migrated cluster, it will be sent a notification with the new location.

For the case of references coming from clusters or proxies held either by the source or destination places, the solution is to convert the references to proxies or local references respectively. Consider again figure 6.3 that shows the configuration before and after migration of cluster C1. Notice how incoming reference from C2 required the creation of a proxy and how proxyY1 in place Y was removed and replaced by a direct reference from C5. Finally, ProxyZ1 remains inconsistent until it receives the notfication of migration. One alternative to update dead proxies like ProxyZ1 consists in defining timers that checks if the links are still valid after a predefined period of time.

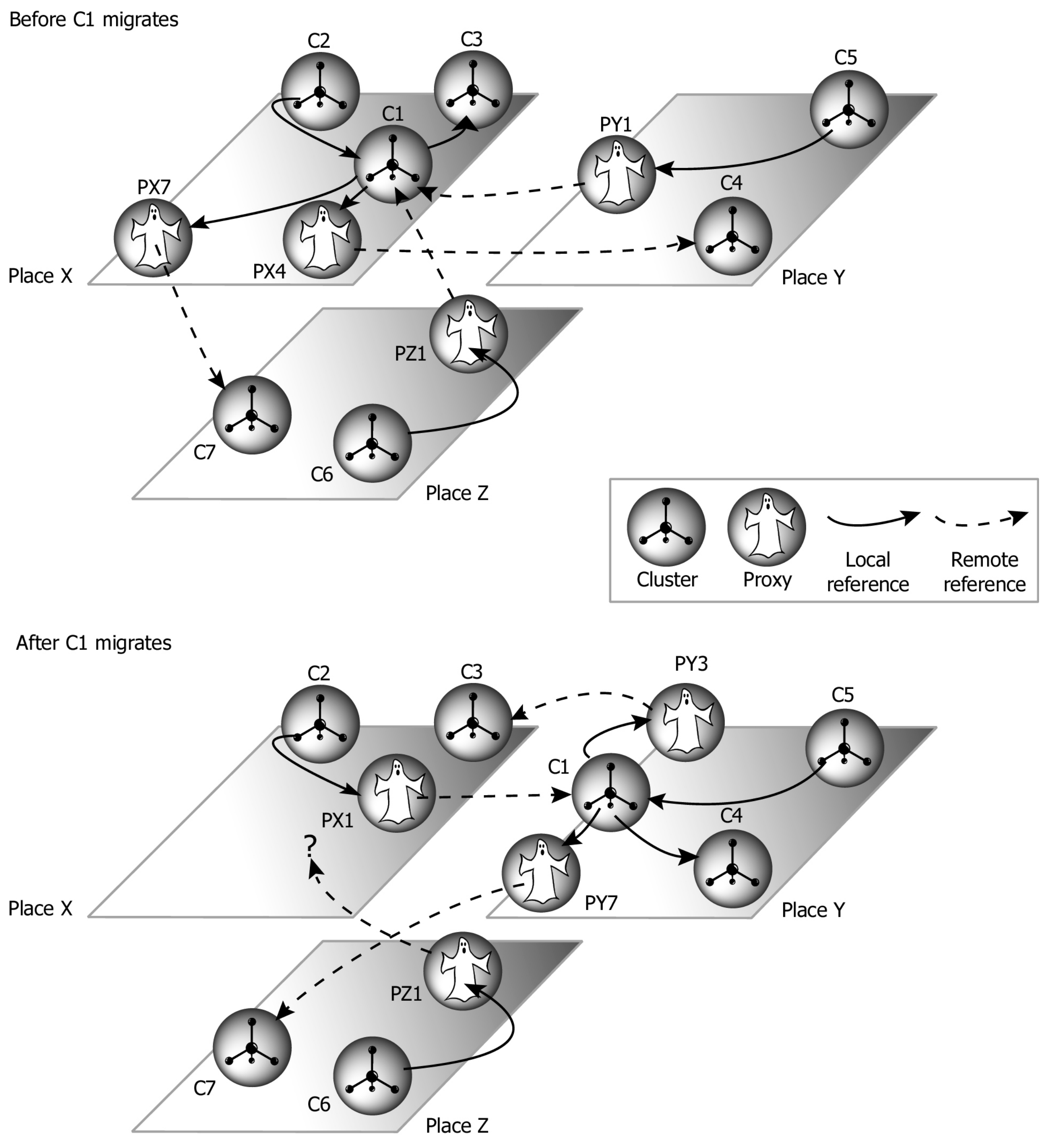


Figure 6.3. An example showing how references are updated after migration

6.2 Implementing the Mobile System Using Reflection

In previous sections we analyzed the technical feasibility of implementing strong mobility in Smalltalk. After making sure that all the necessary aspects are properly reified, we can proceed to provide the guidelines for an implementation in MetaclassTalk. Thus, the goal of this part is to provide some guidelines to implement the ideas of the design discussed previously using metaclasses. MetaclassTalk is all about explicit metaclasses that are used to attach new properties to classes. In our case, we are interesting in defining class properties that provide transparent strong mobility.

The fact that metaobjects can control every aspect of the execution of instance objects allow us to transparently introduce migration. We have decided to give the name of *meta-mobility* to our approach. In general terms, at a given time, the metaobject freezes the execution of the instance and sends it to the destination place. At the destination, the object gets resuscitated and continues its execution. Previous analysis showed the feasibility of doing this.

6.2.1 Implementing Clusters using Metaclasses: MobileClass

We declare a new metaclass named MobileClass that provides all the necessary infrastructure for strong mobility. In this way, any class that is intended to be a cluster should be declared as an instance of MobileClass. More specifically, we requiere that just the class of the facade object be declared as instance of MobileClass; classes of objects living within the frontier can be declared as conventional Smalltalk classes.

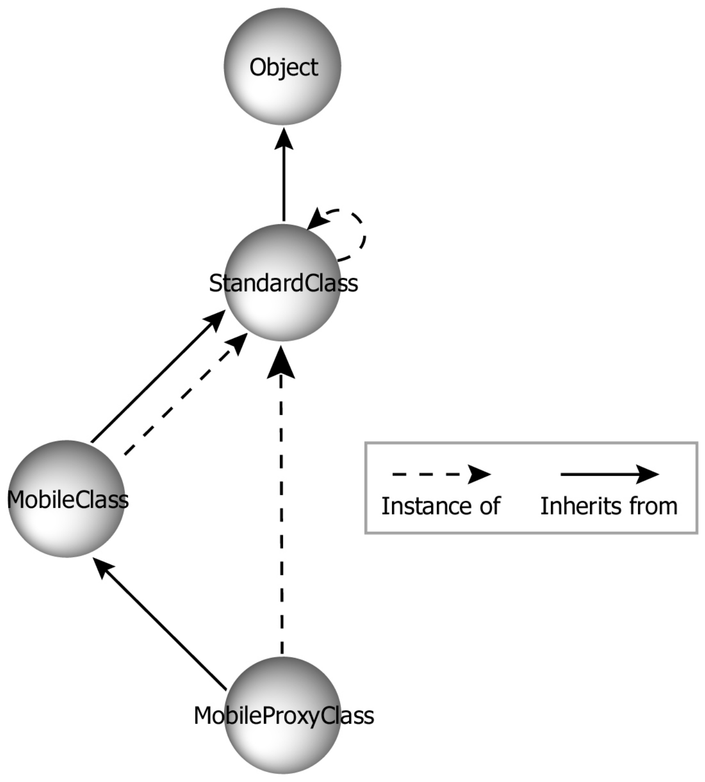


Figure 6.4 MobileClass is the metaclass that provides the funtionality for mobile computation

The core functionality of MobileClass is provided by: *a)* a set of hook point methods that to provide the behavior of classes as metaobjects that control mobility; and *b)* and internal state that customizes the behavior of metaobjects. The hook point methods are intercept the execution of clusters and allow to deal with different issues concerning mobility—migration, incoming and outgoing references and so on. The idea is to identify the main aspects involved in mobility and define a hook point method for each of them, in the same way that MetaclassTalk provides a MOP to control object interaction. On the other hand, the state is an instance of one of very specific state classes that customize the behavior of the metaobject of a cluster.

6.2.2 States of Metaobjects Implementing Mobility

During its lifecycle, a cluster goes through different states. We have identified the following:

- Running: the cluster is executing at some place. The most likely scenario is that of a cluster interacting locally with other clusters available at the same place. Remote interaction is provided transparently through proxies.

- Departing: the cluster is literally frozen while its internal state and the process it was executing in are being externalized to a stream.

- In transit: the data stream that encodes the cluster and its execution is being transfer to a destination place

- Arriving: the cluster has just arrived at the destination and it is being resuscitated. Its internal state is being internalized. The process it was running in is being recreated. When this reconstruction finishes, the process will be resumed and the metaobject of the cluster will make the transition to *running*.

Our approach consists in dealing with the previous states of a cluster transparently from the meta-level. At some extent, we are moving the states from the cluster to the metaobjects. The concept of states for metaobjects is based on the State Pattern [GHJV95]. *"[The State pattern] allows an object to alter its behavior when its internal state changes. The object will appear to change its class"* [GHJV95]. In this way, our approach for the implementation, attaches a state to each metaobject that indicates exactly how to control the execution of a cluster. We define a hierarchy of five classes to define states: ClusterState, RunningCluster, DepartingCluster, InTransitCluster, ArrivingCluster and ProxyCluster. The first one is an abstract class that defines the interface implemented by the others that are declared as its concrete subclasses. The idea with meta-states is to use them as a double-dispatching mechanism [GHJV95] that tells the metaobject how to control the cluster during its execution and upon migration. As all clusters instantiated from the same class share the same metaobject, the meta-state is associated to the cluster; more specifically stored inside the internal structure of the cluster. However, the state is available only for the metaobject and thus it is completely invisible for the cluster. In this way, we simulate the change of the metaobject of the cluster by changing just its state.

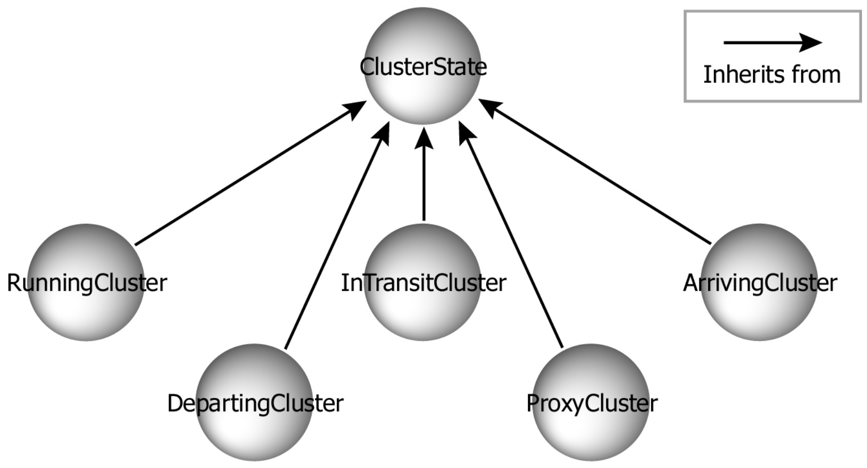


Figure 6.5 Class hierrachy of meta-states

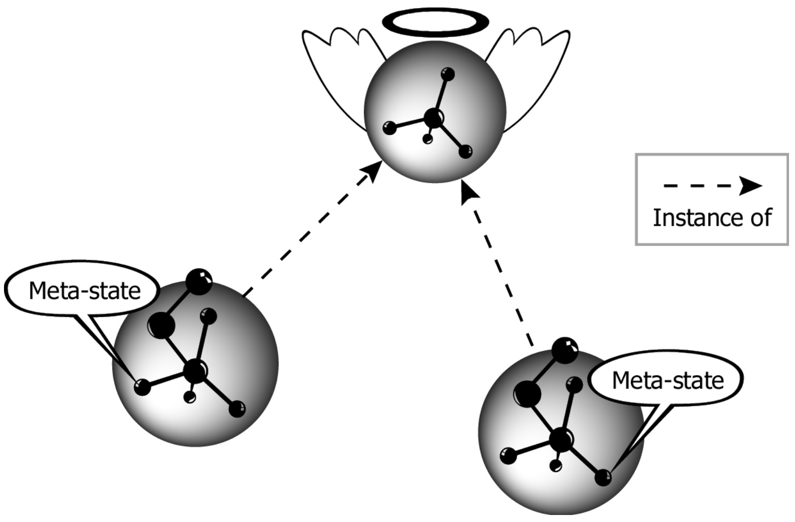


Figure 6.6 Meta-states are kept inside objects but are only visible to metaobjects

6.2.3 Collaborations

According to the state pattern, because all state-specific code lives in a ClusterState subclass, new states and transitions can be added easily by defining new subclasses. In order to control the execution of the cluster, metaobjects delegate on the meta-state the responsibility of deciding what to do. When this happen, meta-states turn around and send the metaobject an specific message. This technique of method invocation is very well-known under the name of double-dispatch [GHJV95]. As an example consider the following fragments of code:

```
MobileClass >>
receive: selector from: sender to: receiver arguments: args superSend: superFlag originClass: originCl

^(self atIV: #metaState of: sender)
        receive: selector
        from: sender
        to: receiver
        arguments: args
        superSend: superFlag
        originClass: originCl


InTransitCluster >>
receive: selector from: sender to: receiver arguments: args superSend: superFlag originClass: originCl

^sender metaobject
        receiveInTransist: selector
        from: sender
        to: receiver
        arguments: args
        superSend: superFlag
        originClass: originCl
```

In this way, all the behavior to control the execution of the clusters is centralized on the metaobject. States simply collaborate performing the proper method invocation.

6.2.4 How Mobility is Provided

Initially, the meta-state associated to a cluster is an instance of RunningCluster. This is the normal state for any cluster living in a place. When a cluster must be migrated, the meta-state is changed to an instance of DepartingCluster. This state instructs the metaobject to freeze the execution of the cluster and begin the externalization of the process to a stream. When this is done, the meta-state is automatically upgraded to InTransitCluster, which instructs the metaobject to perform the transfer of the data stream to the destination place. The destination place receives the stream and finds a new metaobject which performs the revitalization of the cluster. As soon as the object is back to life, an ArrivingCluster state is attached to it which tells the metaobject what to do to put the cluster back to work. When the cluster resumes it execution, the corresponding meta-state is set to RunningCluster again.

There is one important issue to consider is the treatment of incoming requests sent to the cluster while it is being migrated. Our approach allows to handle also this from the meta-level. Thus, meta-states again customize the behavior of metaobjects to handle incomming requests. If the state is DepartingCluster, the metaobject will put all the requests in a queue that will be sent along with the cluster to its destination. If the state is InTransistCluster, incoming requests are not sent along with the cluster. Rather, a notification is delivered back to the senders with the new location of the cluster. When a state is ArrivingCluster—at the destination place—the messages migrated along with the cluster—in the stream—and put in the cluster's queue of incoming request.

6.2.4.1 Migration Policies

There can be many policies that decide when an cluster should migrate. Our approach leaves the treatment of this open to the implementation. In any case, migration can happen either explicitly or implictly. In the former case, the cluster itself decides when to move. To handle this, we need to ensure that: *a)* places are be reified and made available at the level of clusters and *b)* metaobjects are able to intercept the migration requests. Explicit migration encourages the idea of cluster as proactive entities.

On the other hand, with policies of implicit migration, metaobjects themselves decide when clusters are to be migrated. Migration is thus completely transparent at the base level. One policy of implicit migration consists, for example, in migrating the cluster everytime is needs to interact with remote clusters. In this case, places do not need to be reified at the base-level.

6.2.5 Reference Management

During the design we introduced proxies to handle references to external clusters. After migration, we need a way to update the incoming references pointing at the source so that the point at the cluster in its new destination. Our solution consists in installing a proxy at the source location to capture all the requests arriving from old incoming references. This proxy sends a message back to the senders indicating the new location of the cluster. Thus, this scheme gradually updates the references under demand.

In order to install the functionality of such a proxy at the source location, all we need to do is: *a)* leave the facade object at the source location, as external references point at it; *b)* change the state of the metaobject of the cluster so that it behaves like a proxy—ProxyCluster. The mission of a metaobject working as a proxy consists in *a)* send notifications of migration to every remote cluster that sends request to the cluster and *b)* instruct the facade object of the cluster still allocated to release any reference to the internal objects and resources. This latter point is particularly as it deals with the deallocation of the cluster by the garbage collector.

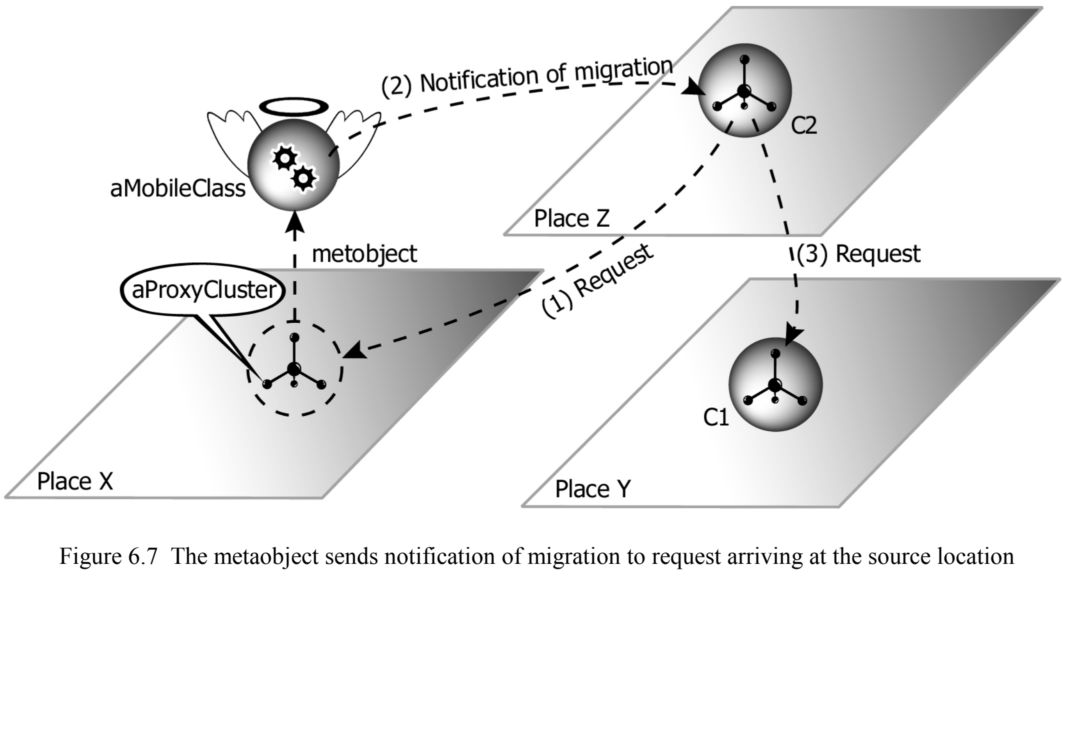


Figure 6.7 The metaobject sends notification of migration to request arriving at the source location

6.2.6 Code Mobility

One important issue to take into account with code mobility is the conflicts that may arise if the code—in the form of classes—is already available at the destination. Classes in Smalltalk are much more than a set of methods; in fact they are objects that include an internal state and references to other objects. So class migration is not a trivial problem. As an alternative, our approach consists in providing the access to classes through proxies. In this way, everytime a cluster moves, a proxy is installed at its destination that provides the access to its class at the place where it was originally instantiated.

The functionality of class proxies is provided by the metaobject of the cluster. We introduce an extra metaclass, MobileProxyClass, subclass of MobileClass, which redefines the method lookup so that it can be performed remotely. At least one instance (class) of MobileProxyClass should be available in every place. This class is the one that gets effectively instanciated for clusters arriving in a new location. The additional information about the remote class that provides the behavior and internal definition of the cluster is encapsulated in the state of the metaobject.

As a side effect, the use of proxies to handle class definitions introduces an extra level of complexity to the serialization techniques, as we have to reinstanciate objects without their class being present locally. The instanciation process in Smalltalk returns a reference to an object allocated in the same address space where the corresponding class is installed. Thus we need to customize the serialization techniques we used. For example, the externalization process must be extended to encode some extra information like the name of the class that effectively will get instanciated at the destination.

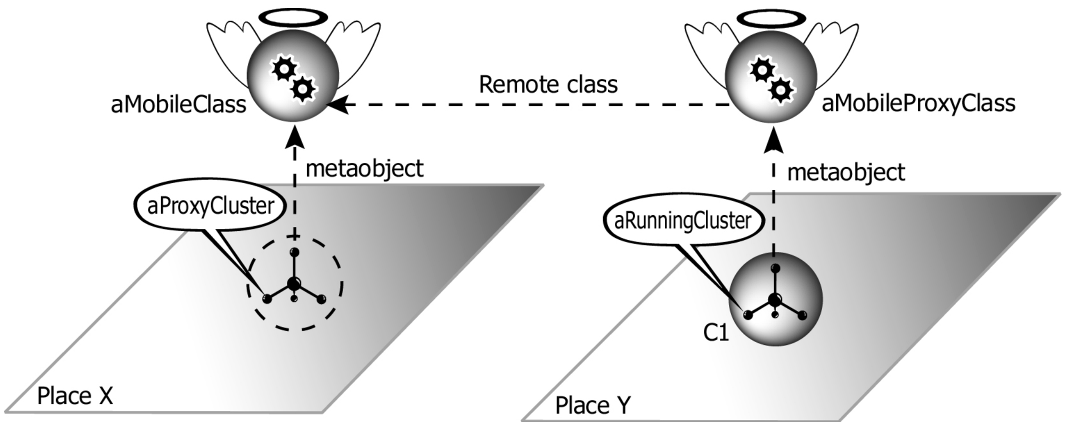


Figure 6.8 A proxy is used to access the class that provides the definition of the cluster

6.2.7 Migration of the State of Execution

Previous sections showed the technical feasibility of migrating a process to continue its execution at remote location. We have already identified and discussed three stages to achieve this: *a)* freeze the process and externalize its state to a stream; *b)* send the data stream to the destination and *c)* internalize the object from the stream and resume its execution. The fact that Smalltalk reifies runtime executions as objects—Processes and contexts—is the key that enables to achieve all this. On the other hand, there is also a framework to externalize and internalize graphs of objects to and from a stream. For our purposes, we need to extend that functionality to deal with externalization—and internalization—of references to global objects that should not be migrated.

Although migration is controlled from the meta-level, we do not think it is a good idea to overload the behavior of metaobjects with those responsibilities. Rather, we would prefer to have the migration facilities as a set of services provided by a set of external objects that collaborate with metaobjects. This approach ensures that migration policies (defined by metaobjects) and migration mechanisms (provided by external collaborators) can evolve independently. As an example, we propose to introduce two extra classes, Externalizer and Internalizer as subclasses of ReferenceStream.

- Externalizer: collaborates with metaobjects to externalize a Process to an stream. In this way, given a Process object, an Externalizer returns a stream with persistent representation of the Process. Its responsibilities include the management of references to global objects. Externalizers also encode the necessary information to re-instatiate clusters when the class is not available at the destination.

- Internalizer: collaborates with metaobjects to internalize a Process from a data stream. In this way, given a data stream, an Internalizer returns an instance of Process ready to be resumed. Its responsibilities include those of restoring references to global objects. Internalizers are also capable of re-instanciating clusters as instances of other classes, different from the original one.

6.3 Summary

In this chapter we presented the major guidelines to design a platform that provides strong mobility. We also discussed an implementation on top of MetaclassTalk.

Conclusions and Future Work

Summary

Strong mobility involves the migration of the runtime image of a software component as a whole, including its execution state. In this dissertation we have illustrated the feasibility of providing *transparent strong mobility* using techniques of *reflective programming* in the context of object-orientation.

First, we studied the advantages of using reflection as a homogeneous technique to open up the internal implementation of a system. The important innovaton of reflective programming languages is that they allow to control and adjust at runtime the internal structures and the execution of the programs that model the application domain. This is particularly important as it enhances the flexibility and adaptability of software systems. Although reflection has not been yet widely accepted, it can be said that the evolution of programming languages tends towards a broader use of reflective facilities.

The next step was to provide an implementation of the MetaclassTalk framework in Squeak—a free available implementation of Smalltalk. MetaclassTalk extends the reflective features of Smalltalk with behavioral reflection, allowing to handle the semantics of inter-object communication. In this sense, four stages are clearly reified: message sending, message reception, method lookup and method application. MetaclassTalk also controls the access to the internal structure of instance objects. The implementation of MetaclassTalk in Squeak differes in many aspects from the previous implementation, mostly because of the differences inherent to each Smalltalk dialect and our commitment to provide an implementation completely independent of NeoClasstalk.

The following stage concerned the study of mobile computation, which refers to the notion that a computation starting at some network node may continue execution at some other network node. The major contribution of mobility is that is defines a new paradigm for software development.

Our focus then moved in the direction of strong mobility. We contributed with the requirements of an infrastructure that gives support for strong mobility and studied into deep detail the reflective facilities available in Smalltalk to implement it. We claim that reflective facilities already available in Smalltalk provide the basis on top of which we can implement strong mobility. Smalltalk refies as objects all the necessary elements involved in migration, like processes, execution context and so on.

The final stage of our work presents some design choices to develop a platform with facilities for strong mobility. As our goal was to induce *transparent* mobility, we discussed some of the issues arising when migration is controlled and performed using reflection—as a separated concern. We provided solutions to these issues based on MetaclassTalk.

Future Work

Some possible directions of future work are analyzed in this section.

• Mobility and Reflection

This dissertation provided a complete analysis of the reflective features available in Smalltalk that need to be taken into account when dealing with strong mobility. We also provided the main guidelines for implementating strong mobility using metaobjects in MetaclassTalk. In this way, mobility would be transparently provided and controlled from the meta-level, without interfering with the code of the application domain.

However, a complete implementation would arise many technical issues that need to be addressed. For example, the development of a framework to deal remote interaction using proxies, provide a naming service, define policies to deal with parameters sent among remote clusters and so on.

• The MetaclassTalk Language

The MetaclassTalk language offers a MOP that allows to control inter-object communication and the access to the internal structure of instance objects. One possible evolution path could be to extend the MOP to support also some control over the set of incoming references to an object. More specifically, right now, MetaclassTalk provides a hook point method that controls the assigment of instance variables. That is, given an object that includes an instance variable var we can control the asignment of a value aValue to var. However, no mechanism is provided to give aValue a chance of deciding how it wants to be assigned to var. This is because referenced objects have no control over their incoming references, rather they can control only their outgoing references. It would be nice that aValue was asked how it wants to be assigned into a variable, so that it can have a chance to return a wrapper of itself or another object representing it—a proxy. Notice that this approach is sending the control of the assignment operation literally to the *other endpoint* of the reference link. We believe that this reification would introduce new semantics to the way in which proxies and incoming references are treated in our model. One possible application of this would be the development of automatic mechanisms to de-reference remote objects.

• Metaclasses vs. Metaobjects

On the other hand, MetaclassTalk's object model is based on the use of classes as metaobjects. Another path of evolution could be to split classes and metaobjects as two separated entities. The strong advantages of this is that: *a)* it encourages a clear and sharp separation of concerns; that is, classes would be used to model *only* the application domain while metaobjects would control the execution and internal representation of instances—which has nothing to do with the domain itself; and *b)* allows classes and metaobjects evolve independently. Some work has been done before in this direction [Mae87]. However, separation is not always desirable and also has some drawbacks, mostly because metaobjects heavily rely on classes to perform their tasks. For example: at creation time, a link must be created between the instance object and its metaobject. Who creates such a link? It is a responsibility for the metaobject, but at that stage, the metaobject is not available yet. Another drawback concerns the administration of the space allocated for instances. While the definition is provided by the class, the access to the internal structure is controlled by metaobjects. If the resposibilities are separated in to different entities, we miss the possibility of implementing some optimization like allocation under demand.

• Mobility, Reflection and Security

While reflection opens up and puts some light on the dark side of system implementations, mobility encourages the migration of self-contained entities that come and go from different places. The merge of the two somehow represents an *explosive combination* from the secutiry point of view. In fact, the main obstacle to the acceptance of mobile computation for commercial application is the issue of security. It could be very interesting to explore the advantages of providing security using reflection

Appendix A

Say Hello to MetaclassTalk—Installation Procedure

Requirements

The installation of MetaclassTalk for Squeak is a procedure of seven steps. The fact that StandardClass, MetaclassTalk's kernel, is an instace of itself, requires some extra actions if compared to the installation of a normal Smalltalk application. In fact, to make StandardClass an instance of itself requires an extra primitive in the virtual machine, to change the referece (pointer) of an object to its class. Bret Pinkney integrated such primitive [BrP] in the virtual machine of Squeak v2.7 for Windows. We require such a virtual machine to perform the installation.

In order to install MetaclassTalk for Squeak, the following files should be available:

- '1 new primitive.st'
- '2 install BootClass.st'
- '3 create StandardClass.st'
- '4 extended classes.st'
- '5 extra classes.st'
- '6 StandardClass methods.st'
- '7 change metaclass.st'

Installation

Please, check that you are working with the extended virtual machine before proceeding.

Step 1 Installing the primitive

If your image includes the installation of the primitive invocation in the Interpreter, move to the next step. Otherwise, if your image is a completelly new one, evaluate the following in a workspace:

```
(FileStream oldFileNamed: '1 new primitive.st') fileIn
```

Step 2 Boostrapping

The fact that StandardClass is an instance of itself, implies a problem of vicious circularity at creation time. This is a classic chicken-and-egg problem: StandardClass cannot be created until its class exists and this class needs to be an instance of itself. The issues concerning this particular type of circularity are known as *bootstrapping issues* [KdRB91]. They are involved with how to get the system up and running. Fortunately, once the system is installed, the problem evaporates.

The solution to this requires a two-step installation technique using an additional class, BootClass, which temporarily acts as the class of StandardClass. Both classes define almost the same protocol. Later, the reference from StandardClass to BootClass will be changed to point at itself. In order to install BootClass, evaluate:

```
(FileStream oldFileNamed: '2 install BootClass.st') fileIn
```

Step 3 Creating StandardClass

```
BootClass new superclass: Object;
    instanceVariables:  (Array with: 'superclass' with: 'methodDict' with: 'format' with: 'body');
    installAtName: #StandardClass
    category: #'MetaclassTalk Kernel'
```

The previous expression creates StandardClass as an instance of BootClass. In order to do this, evaluate:

```
(FileStream oldFileNamed: '3 create StandardClass.st') fileIn
```

Step 4 Adding New Methods to Conventional Smalltalk Classes

Some of the methods available in conventional Squeak classes were extended or minimally changed to support the integration of MetaclassTalk, as discussed in Section 3.3.2. The evaluation of the following expression will upload such changes into Squeak.

```
(FileStream oldFileNamed: '4 extended classes.st') fileIn
```

Step 5 Installing New Classes

The MetaclassTalk framework is constituted by some extra classes that redefine and extend the default behavior of the Smalltalk system. To install those classes, evaluate the coming expression:

```
(FileStream oldFileNamed: '5 extra classes.st') fileIn
```

Step 6 Installing the Protocol of StandardClass

In (step 3) we created StandardClass as an instance of BootClass, but no protocol for it was installed. We will procced to do it now, as all the necessary infraestructure is available. Evaluate the following:

```
(FileStream oldFileNamed: '6 StandardClass methods.st') fileIn
```

Step 7 Making StandardClass an Instance of Itself

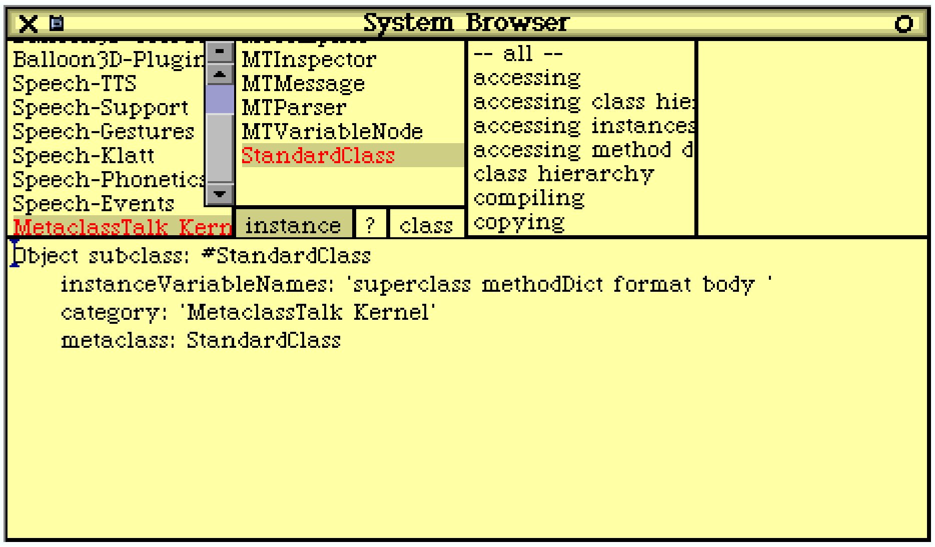
The second part of the bootstrapping procedure consists in making StandardClass its own metaobject. In Smalltalk, the expression to do this is:

```
StandardClass class: StandardClass
```

You can either type and evaluate the previous expression, or file it in from the following:

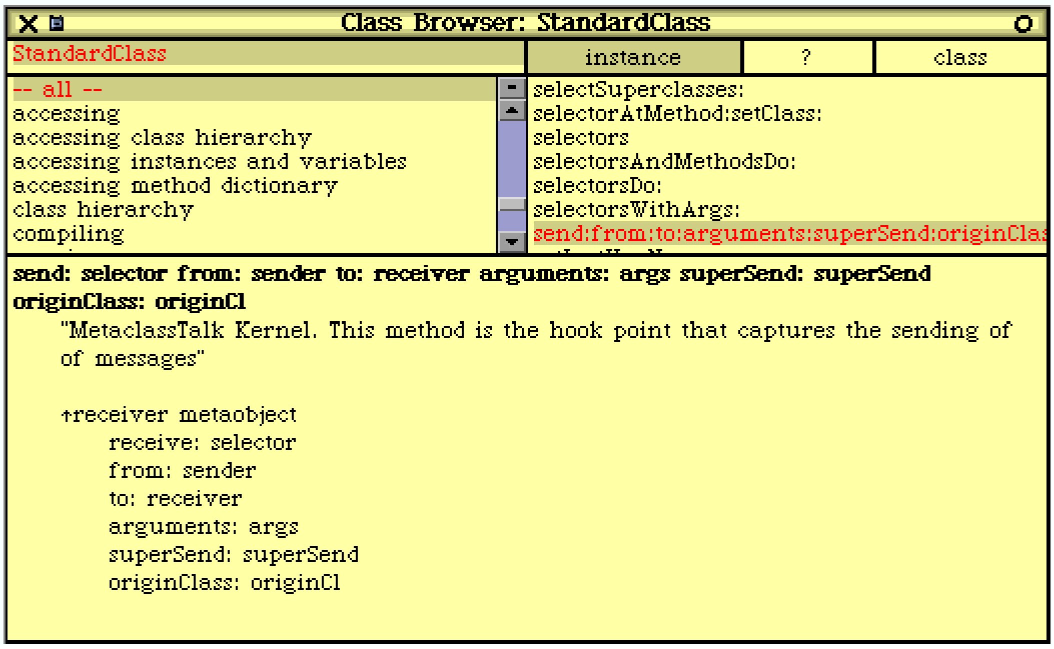
(FileStream oldFileNamed: '7 change metaclass.st') fileIn

The installation is finished. Take your time and say hello to MetaclassTalk for Squeak!
Save the image and replace the extended virtual machine with a normal one (optional). Figure 3.2 shows a browser in Squeak with the definition of StandardClass. Figure 3.3 shows the source code of the method StandardClass >> send:from:to:arguments:superSend: originClass:



```
Object subclass: #StandardClass
    instanceVariableNames: 'superclass methodDict format body '
    category: 'MetaclassTalk Kernel'
    metaclass: StandardClass
```

Figure 3.2 Definition of StandardClass inMetaclassTalk for Squeak


Class Browser: StandardClass
StandardClass
instance
?
class
-- all --
accessing
accessing class hierarchy
accessing instances and variables
accessing method dictionary
class hierarchy
compiling
selectSuperclasses:
selectorAtMethod:setClass:
selectors
selectorsAndMethodsDo:
selectorsDo:
selectorsWithArgs:
send:from:to:arguments:superSend:originClas

```
send: selector from: sender to: receiver arguments: args superSend: superSend
originClass: originCl
    "MetaclassTalk Kernel. This method is the hook point that captures the sending of
    of messages"

    ↑receiver metaobject
        receive: selector
        from: sender
        to: receiver
        arguments: args
        superSend: superSend
        originClass: originCl
```

Figure 3.3 Source code of the method send:from:to:arguments:superSend:originClass:

References

[BDLS00] Noury Bouraqadi-Saâdani, Rémi Douence, Thomas Ledoux and Mario Südholt. Un mòdele de mobilité forte pour Java. Technical Report École des Mines de Nantes – Dépt Informatique, France. July 2000.

[BrP] Bret Pinkey, Extension to the VM of Squeak available at: http://sites.netscape.net/pinkles/neosqueak

[BFJR98] J. Brant, B. Foote, R. Johnson and D. Roberts, Wrappers to the Rescue, *Proceedings of ECOOP ' 98*

[BF90] Brian Foote, Object-Oriented Reflective Metalevel Architectures: Pyrite or Panacea? A Position Paper for the *ECOOP/OOPSLA '90 Workshop on Reflection and Metalevel Architectures*

[BS99] Noury Bouraqadi-Saâdani, Un MOP Smalltalk pour l'étude de la composition et de la compatibilité des métaclasses. Application à la programmation par aspects. PhD thesis, Université de Nantes, July 1999.

[Car97] Luca Cardelli. Mobile Computation. J. Vitek and C. Tschudin Editors. Mobile Object Systems - Towards the Programmable Internet. Lecture Notes in Computer Science, Vol. 1222, Springer, 1997. pp 3-6

[Coi87] Pierre Cointe. Metaclasses are First Class: the ObjVlisp Model. In *Proceedings of OOPSLA '87*

[Coi90] Pierre Cointe. The ClassTalk System: a Laboratory to Study Reflection in Smalltalk. *In Informal Proceedings of the First Workshop on Reflection and Meta-Level Architectures in OO-Programming, OOSPLA/ECOOP'90*

[DH98] Daniel Hagimont. Modèle à Agents Mobiles, INRIA

[DBW93] R.G. Gabriel, D.G. Bobrow and J.L. White: CLOS in Context – The Shape of the Design Space. In Object Oriented Programming – The CLOS Perspective. MIT Press, 1993

[Fer89] Jacques Ferber: Computational Reflection in Class-based Object Oriented Languages. *Proceedings of OOSPLA '89*

[FPV98] A. Fuggetta, G.P. Picco, G. Vigna. Understanding Code Mobility, *IEEE Transactions on Sftware Engineering,* Vol 24, No. XX, XXXXX, 1998

[GB99] Guy Bernard. Technologie du Code Mobile: éetat de l'art et perspectives, Institut National des Télécommunications,

[GHJV95] E. Gamma, R. helen, R. Johnson and J. Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995

[GR83] A. Goldberg and D. Robson. *Smalltalk-80, The Language and its implementation*. Addison Wesley, Readings, Massachusetts, 1983.

[HH95] Trevor Hopkins, Bernard Horan. *Smalltalk : An Introduction to Application Development Using VisualWorks.* Prentice Hall, 1995

[HL98] D. Hagimont and D. Louvegnies. Javanaise: Distributed Shared Objects for Internet Cooperative Applications. *Proceedings of Middleware '98*

[HVL95] Walter L. Hürsch and Crisitna Videira Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, February 1995

[IKMWK97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace and Alan Kay. Back to the Future. In *Proceedings of OOPSLA '97*

[KdRB91] Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow. *The Art of the Metaobject* Protocols. MIT Press, 1991

[Kic96] Gregor Kiczales, Beyond the Black Box: Open Implementation. January 1996 Issue of IEEE Software.

[Lan98] Danny B. Lange, Mobile Objects and Mobile Agents: The future of Distributed Computing? *Proceedings of ECOOP ' 98*

[LO98] Lange, D.B. and Oshima, M.: Programming and Deploying Java Mobile Agents with Aglets, Addison-Wesley, 1998

[LP90] Wilf R. Lalonde and John R. Pugh. Inside Smalltalk (Vol 1). Prentice-Hall International Editions, Englewood Cliffs, New Jersey, 1990.

[Mae87] Pattie Maes: Concepts and Experiments in Computational Reflection. *Proceedings of OOSPLA '87*

[McA95] Jeff McAffer, Meta-level Programming with CodA, *Proceedings of ECOOP ' 95*

[MJD96] J. Malefant, M. Jacques and F. N Demers. A Tutorial on Behavioral Reflection and its Implementation. *Proceedings of Reflection '96, San Francisco*, USA

[OHE96] R. Orfali, D Hankey, J. Edwards. *The Essential Distributed Objects Survival Guide. John Wiley & Sons. 1996*

[OMG] Object Management Group, on-line documentation at http://www.omg.org

[Riv96] Fred Rivard: Smalltalk: a Reflective Language. *Proceedings of Reflection '96, San Francisco,* USA

[Riv97] Fred Rivard, Évolution du Comportement des Objets dans les Langages à Classes Réflexifs Thèse de doctorat, Université de Nantes, June 1997

[Smi82] Smith B., Reflection and Semantics in a Procedural Language. Massachusetts Institute Of Technology. Laboratory for Computer Science. Technical report 272. Cambridge, Massachusetts.

[SK] Sacha Krakowiak. Code Mobile, Principles et Mise en Œuvre, Université J. Fourier, Labo Sirac (INPG-INRIA-UJF)

[SUN] Sun Microsystems, Java on-line documentation, http://java.sun.com/

[VB00] Vassili Bykov. *The Hitch Hiker's Guide to the Smalltalk Compiler*. Smalltalk Chronicles, vol 2, No 1, March 2000

[Whi96] Jim White, *Mobile Agents White Paper*, General Magic, 1996