Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
and
Universidad de Chile - Chile
2000

Reflex

– A Reflective System for Java –

Application to Flexible Resource Management in Mobile Object Systems

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

By: Eric Tanter

Promotor: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promotor: Dr. José Piquer (Universidad de Chile)

Acknowledgments

I would like to take this opportunity to thank the people who helped to make this work possible:

First of all, thanks to all the organizers of the EMOOSE project for their vision and commitment. The EMOOSE project was a fantastic experience, and I do hope it will be reconducted over years.

Thanks to my promotor Prof. Dr. Theo D'Hondt for taking all EMOOSE students under his wings. Also thanks to my co-promotor Dr. José Piquer, for suggesting a subject, and allowing me to deviate from it and finally work on what I wanted. I especially appreciate the freedom I was given. Thanks to Dr. Luis Mateu, for the many technical discussions we have had, allowing me to confront and therefore improve my ideas.

I also would like to thank Dr. Jacques Noye for giving some of its precious time to proofread this thesis and for the highly helpful comments he gave me.

Thanks to the people of the supporting institutions, the Ecole des Mines de Nantes and the Universidad de Chile for helping me whenever they could, in particular during my stay in Chile.

Last, but not least, many thanks to my parents for fully supporting me during this year, and for their understanding of my personal choices.

Contents

1 Introduction **1**
- 1.1 Goals . . . 1
- 1.2 Overview . . . 2
- 1.3 Note for the Non-Technical Reader . . . 3

2 Concepts **4**
- 2.1 Code Mobility . . . 4
  - 2.1.1 Mobile Code Technologies . . . 6
  - 2.1.2 Applications of Code Mobility . . . 11
  - 2.1.3 Code Mobility in Java . . . 13
- 2.2 Meta-Programming and Reflection . . . 14
  - 2.2.1 Principles . . . 14
  - 2.2.2 Applications of Reflection . . . 16
  - 2.2.3 Reflection in Java . . . 16
- 2.3 Reflection for Code Mobility in Java . . . 16
- 2.4 Summary . . . 17

3 Reflex **18**
- 3.1 Introduction . . . 18
- 3.2 Motivation and Objectives . . . 19
  - 3.2.1 Transparent Type-Compatibility . . . 20
  - 3.2.2 Creation of Reflective Objects . . . 22
  - 3.2.3 A Framework for Metaobjects Composition . . . 23
  - 3.2.4 An extensible MOP . . . 25
- 3.3 Architecture of Reflex . . . 26
  - 3.3.1 The Javassist Library . . . 26
  - 3.3.2 Building Reflective Classes . . . 31
  - 3.3.3 The Meta-level Architecture . . . 33
  - 3.3.4 The Metaobject Protocol . . . 38
  - 3.3.5 The Reflex Public Interface . . . 42
- 3.4 Summary . . . 43

4 Reflex for Mobile Object Systems **44**
4.1 Introduction 44
4.2 Interfacing with Mobile Object Systems 45
4.2.1 Kinds of Serialization 46
4.2.2 Stream Identifiers 46
4.2.3 The StreamMetaobject Class 47
4.3 Serialization Awareness and Control 48
4.3.1 Serialization in Java 48
4.3.2 Serialization Awareness 48
4.3.3 Serialization Control 49
4.3.4 Serialization Wrapper 50
4.3.5 Extension of the Reflex Framework 53
4.3.6 Advanced Serialization Issue 55
4.4 Summary 58

5 Metaobjects for Resource Management **59**
5.1 Introduction 59
5.2 The Rebinding Policy 60
5.2.1 Design and Implementation 61
5.2.2 Examples 61
5.3 The Network Reference Policy 64
5.3.1 Design and Implementation 64
5.3.2 Examples 66
5.4 Summary 68

6 Evaluation **69**
6.1 Achievements 69
6.2 Performance 70
6.3 Limitations 71
6.3.1 Final classes 71
6.3.2 Final methods 72
6.3.3 Composition of metaobjects 72

7 Conclusions and Perspectives **74**
7.1 Future Work 74
7.2 Perspectives 74
7.3 Conclusions 76

A Paradigms of Distributed Computing **77**
A.1 Client-Server 77
A.2 Remote Evaluation 77
A.3 Code on Demand 78
A.4 Mobile Agent 78
A.4.1 What's a Software Agent? 78

A.4.2 The Paradigm 79
A.4.3 Drawbacks of the Mobile Agent Paradigm 79
A.5 Summary 81

B EzAgent **82**
B.1 Concepts 82
B.2 The ezagent.EzAgent class 82
B.3 The ezagent.EzPlace interface 83
B.4 The ezagent.EzPlaceImpl class 85
B.5 Summary 85

C Resource Manager for Java **86**
C.1 The java.lang.ResourceManager class 86
C.2 The new java.lang.System class 88
C.3 Summary 89

List of Figures

2.1 The internal structure of an executing unit. 7
2.2 Data space management mechanisms. 10

3.1 How Javassist makes a class reflective. 29
3.2 Metaobjects and their dependencies—(a) a root metaobject alone. (b) non-cooperative metaobject added. (c) cooperative metaobject added. 34
3.3 UML diagram of the metaobject composition framework . . . 36

4.1 Reflective objects and migration—(a) A simple configuration before migration of the mobile agent. (b) After migration has taken place. The question marks indicate the specification points for implementing different semantics. 45
4.2 Principle of a serialization wrapper. 50
4.3 Creating a transmitting a blank object—the necessity of switching the metaobject link. 52

5.1 Design of the rebinding policy—(1) Lazy-initialization of the resource. (2) The resource is initialized, calls are forwarded to it. 62
5.2 Design of the network reference policy. 65

List of Tables

3.1 Main methods of the Metaobject class of Javassist. 30
3.2 Implicit part of the inter-level MOP. 39
3.3 Explicit part of the inter-level MOP. 40
3.4 Methods of the intra-level MOP. 41
3.5 Extra methods of the intra-level MOP for cooperative metaobjects. 42

4.1 The StreamIdentifier interface. 47
4.2 The SerializableReflexObject interface. 54

B.1 Customization methods of the life-cycle of an EzAgent. . . . 83
B.2 The EzPlace interface. 84

Chapter 1

Introduction

Computer networks are evolving at a fast pace, in particular with the advent of the Internet and intra- and inter-organization networks. Networks are invading our society, may it be at a business level or at a personal level.

There are more and more users of computer networks every day, and we can foresee that this expansion will go on for a while. Indeed, a huge number of small electrical devices will make use of network connection in the future, in houses as well as in companies. It is predictable that soon many persons will carry along with them a small device connected to the Internet (or its successor), using it for a tremendous variety of applications.

This evolution is changing the settings of the computing world, introducing contrasts in technologies, such as wireless networks and high-bandwidth networks. This in turns implies new requirements for software. It should now be adaptable to network characteristics in order to optimize the use of resources, it should be easily updatable, in a distributed manner, since the topology of the network will not be static anymore, etc.

The mobile code distributed paradigm is an emerging approach to address the issues of this new environment. The need for adaptability and flexibility is however still pending, since these days research on mobile code is more focused on issues like security and performance.

In traditional computing, distributed or not, computational reflection is an interesting approach to build open systems using a powerful separation of concerns. Therefore it seems attractive to apply reflection to mobile code in order to achieve the adaptability that mobile programs will need in the near future.

1.1 Goals

The objective of this thesis is to build a reflective system for Java that can be used to achieve flexibility in mobile object systems. Java is a programming language that has raised many expectations in the area of code mobility,

and is widely used in the Internet today. Therefore most current research works on mobility are done using Java.

Though the reflective system we aim at building is targeted at mobile object systems, we want to make it generic enough to be applicable to other domains. The idea is to first build an appropriate reflective system in Java, without any assumption on the application domain. Then, we want to extend this generic system to be suited to mobile object systems.

We want to use this system to address one particular issue of flexibility in mobile object systems: that of resource management policies. Indeed, in current mobile object systems, the resource management policy is chosen in a fixed manner (by-copy, by-reference, by-move, etc.). This fixed choice is however incompatible with the need mobile programs have to be able to adapt themselves to network status. It should be possible to specify which policy should be used for each resource, without entailing tough programming.

Finally, we aim at opening interesting perspectives concerning the use of reflection to make mobile object systems flexible, offering a concrete working product that can be used for experimenting new ideas.

1.2 Overview

The next chapter will introduce a number of concepts about the fields of code mobility and computational reflection. From this presentation will arise the technical motivation of this dissertation, namely reflection for code mobility in Java.

Chapter three will present Reflex, a reflective Java system that we have built during this thesis. The motivation and objectives of Reflex as well as its detailed architecture are exposed.

The fourth chapter will discuss the extensions made to the Reflex system to make it operational in a distributed environment. In particular, the interface between Reflex and mobile object systems is analyzed.

In chapter five we will present two classes of metaobjects that we have developed in order to apply Reflex to resource management policies. Along with these presentations, small practical examples are included.

Chapter six will discuss the evaluation of our work, focusing on the Reflex system: performance issues and current limitations of the system are highlighted.

The final chapter will present our conclusions and suggest some topics for future work and perspectives for further research.

1.3 Note for the Non-Technical Reader

The present report is a master's thesis, thus focusing on research related issues. However, since it should as well play the role of an industrial project report, we have included some applicative elements. Hereby follows an indicative roadmap that should help the non-technical reader select which parts of the thesis are worth reading.

Chapter two is probably the most interesting part for the non-technical reader, since it explains the concepts of the research work carried out, and highlights the potential benefits and application domains of the fields.

The beginning of chapter three is also well-suited to a non-technical reader since it introduces the Reflex system we built, with its motivation and objectives. The last part of the chapter presenting the architecture of the system is much more technical and can therefore be skipped.

The introduction of chapter four presents the ideas behind Reflex for distributed systems, and is thus of interest for the non-technical reader. The following of the chapter keeps on increasing in terms of technical complexity. We believe that at least the beginning of section 4.2 should be read, since it introduces the main idea of the interface between Reflex and mobile object systems. The rest of the chapter could be skipped, in particular section 4.3.6 which goes deep down into details.

We think that chapter five is easy to read and gives a good illustration of the use of our work, in concrete situations. The principles are presented at a high conceptual level, and no implementation details are addressed.

Finally, chapter six and seven are of interest to any non-technical reader since they highlight our own evaluation and conclusions on our work.

We hope that this brief roadmap will help the non-technical reader make his way through this dissertation and get a good view of the research work we have been doing during this thesis.

Chapter 2

Concepts

In this chapter we introduce some necessary concepts related to code mobility and reflection. We first survey the relevance of the research field of code mobility, and then present mobile code technologies and their concrete applications. Then we expose the principles of reflection and meta-programming, emphasizing the benefits of such an approach to solve issues of mobile code technologies. We end up identifying the technical motivation of this dissertation.

2.1 Code Mobility

Computer networks are evolving at a fast pace, and this evolution proceeds along several lines: the *size* of the network is growing rapidly, let it be the Internet or intra- and inter-organization networks. Side effects of this growth is the significant increase of the network traffic, which implies efforts to enhance the *performance* of the communication infrastructure, and network complexity (unpredictable response times, availability, packets loss, etc.).

In turn, the increase in size and performance of computer networks is both the cause and the effect of the pervasive and ubiquitous nature of networks [32]. Indeed, network connectivity has became a basic feature of any computing facility and will probably be so for many products in the consumer electronics market, thus making networks *pervasive*. Recent developments in wireless technologies free the network nodes from the constraint of being placed at a fixed physical location, enabling the advent of so-called *mobile computing*, where users can move together with their hosts across different physical locations and geographical regions. This is networks *ubiquity*.

The increasing availability of easy-to-use technologies like the World Wide Web has implied the creation of new application domains and markets, therefore changing the nature and *role* of networks.

However, all this evolution poses several challenging problems that must

be addressed [32]. The increase in size of networks raises a problem of *scalability*. Wireless connectivity poses even tougher problems [30]. Since network nodes can move and be connected discontinuously, the topology of the network is no longer defined statically. This undermines some of the basic tenets of research in distributed systems, implying the need of adapting to this new scenario. Also, the diffusion of network services and applications to very large segments of our society makes it necessary to increase the *customizability* of services. Finally, the dynamic nature of both the underlying communication infrastructure and the market requirements demand increased *flexibility* and *extensibility*.

Most of the proposed approaches to provide answers to this multifaceted problem try to adapt well-established models and technologies within the new setting, and usually take for granted the traditional client-server architecture. CORBA [36] is an example of such an approach, relying on remote procedure calls. However, those approaches do not ensure the degree of flexibility, customizability and reconfigurability needed to cope with the requirements discussed above.

A different approach comes from the promising research area exploiting the notion of *mobile code*. Code mobility can be defined informally as the capability to dynamically change the bindings between code fragments and the location where they are executed [16]. This is a powerful concept that originated a very interesting range of developments.

However, despite the wide interest in mobile code technology and applications, the research field is quite immature. There is not even a commonly agreed term to qualify the subject of this research: code mobility, mobile code, mobile computations, mobile object systems, or again program mobility are widely used terms, depending on the authors. In [32], the authors achieve a step towards a sound terminological and methodological framework for the field. This section on code mobility is deeply based on this pioneer work.

The main attempt of the latter work is to enlight the confusion about the semantics of mobile code concepts and technologies. There is a clear distinction to be made between technologies, design paradigms and applications domains ([32]):

Mobile code technologies are the languages and systems that provide *mechanisms* enabling and supporting code mobility. They are used by the application programmer in the implementation stage.

Design paradigms are the *architectural styles* that the application designer uses in defining the application architecture. Client-server is a well-known example of design paradigm.

Application domains are *classes of applications* that share the same general goal, *e.g.* distributed information retrieval or electronic commerce.

> The expected benefits of code mobility in a number of application domains is the motivating force behind this research field.

In this thesis we target at improving mobile code technologies by addressing a specific issue of these technologies. Therefore we are not directly concerned by the mobile code paradigms. However, the interested reader can find in Appendix A a short description and comparison of the different paradigms of distributed computing, with a special focus on the emerging mobile agent paradigm.

Similarly, application domains are not of our concern. However, to motivate the necessity of code mobility, the main application domains of code mobility are exposed later in this chapter as well as the expected key benefits.

2.1.1 Mobile Code Technologies

The technical motivation of this dissertation arises from the limitations encountered in mobile code technologies available today. Hereafter we introduce the terminology exposed in [32] to characterize distributed systems making use of code mobility.

Terminology

In technologies supporting code mobility, the structure of the underlying computer network is not hidden from the programmer like in traditional distributed system, rather it is made manifest. A *Computational Environment* (hereafter, CE), retain the "identity" of the host where it is located. The purpose of the CE is to provide applications with the capability to dynamically relocate their components on different hosts. It therefore handles the relocation of code, and possibly of state, of the hosted software components.

Components hosted by the CE can be separated in two categories: *executing units* (EUs) and *resources*. Executing units represent sequential flows of computation, *e.g.* single-threaded processes or individual threads of a multi-threaded process. Resources represent entities that can be shared among multiple EUs, such as a file in a file system, or, what is of interest to us, an object shared by threads in a multi-threaded object-oriented language (like Java).

EUs are modeled as the composition of a *code segment*, which provides the static description for the behavior of a computation, and a *state* composed of a *data space* and an *execution state*, as illustrated on Figure 2.1. The data space is the set of references to resources that can be accessed by the EU. These resources are not necessarily co-located with the EU on the same CE, as we will see later. The execution state contains private

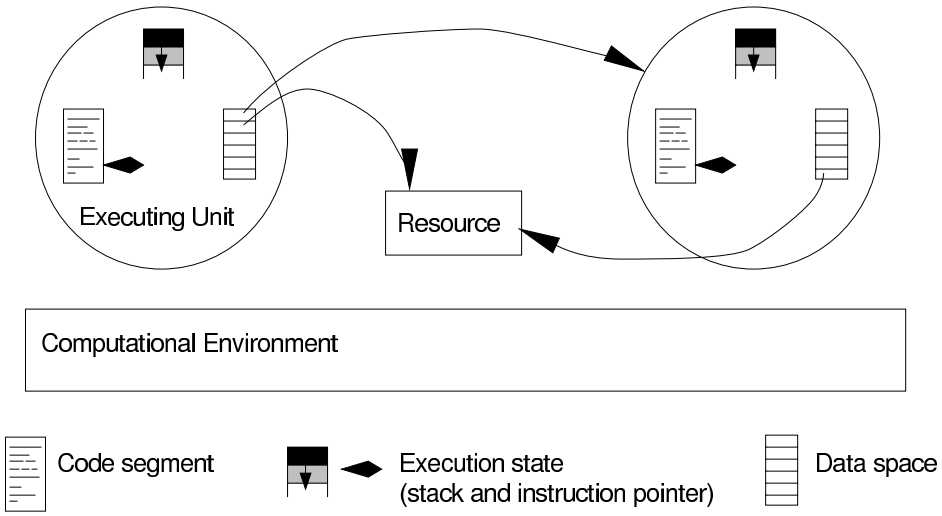

Figure 2.1: The internal structure of an executing unit.

data that cannot be shared, as well as control information related to the EU state, such as the call stack and the instruction pointer.

Mobility mechanisms

In conventional systems, each EU is bound to a single CE for its entire lifetime. In Mobile Code Systems (MCSs), the code segment, the execution state, and the data space of an EU can be relocated to a different CE. The portion of an EU that needs to be moved is determined by composing *orthogonal* mechanisms:

- Mechanisms supporting mobility of code and execution state;
- Mechanisms for data space management.

Though our work aims at providing an enhanced mechanism for data space management for Java MCSs, we introduce briefly the different mechanisms for code and execution state mobility.

Code and execution state mobility

Existing MCSs offer two forms of mobility, characterized by the EU constituents that can be migrated:

Strong mobility is the ability of an MCS to allow migration of both the code and the execution state of an EU to a different CE;

Weak mobility is the ability of an MCS to allow code transfer across different CEs; code may be accompanied by some initialization data, but no migration of execution state is involved.

In [32], the authors propose a detailed classification of code and execution state mobility mechanisms that goes beyond the two forms presented here. Strong mobility mechanisms can support either *migration* or *remote cloning*, both either in a *proactive* or *reactive* manner. Weak mobility mechanisms provide either *code shipping* or *code fetching*. The moved code can either be *stand-alone code* or *code fragment*, and the mechanism can operate either in a *synchronous* or *asynchronous* way. In asynchronous mechanisms, the actual execution of the code transfer can occur either in an *immediate* or *deferred* fashion. We refer the interested reader to the paper for more information on the topic.

Data space management

Upon migration of an EU to a new CE, its data space, *i.e.*, the set of bindings to resources accessible by the EU, must be rearranged. The way it is rearranged depends on the nature of the resources involved, the type of binding to such resources, as well as on the requirements posed by the application. Fuggeta et al. model resources as a triple $Resource = \{I, V, T\}$, where I is a unique identifier, V is the value of the resource, and T is its type, which determines the structure of the information contained as well as its interface. The nature of the resource determines the possible data space management mechanisms. They distinguish three kind of resources:

- A *free transferable* resource can be migrated over the network (*e.g.* a data file);
- A *fixed transferable* resource could be migrated over the network, but it is not the case (*e.g.* a huge or crucial data file);
- A *fixed not transferable* cannot be migrated over the network (*e.g.* an OS printer handle).

Resources can be bound to an EU through three forms of binding:

- The strongest form is *by identifier*: the EU requires that, at any moment, it must be bound to a given uniquely identified resource. The EU is interested in the *identity* of the resource.
- A binding established *by value* declares that, at any moment, the resource must be compliant with a given type and its value cannot change as a consequence of migration. The EU is interested in the *content* of the resource.

- The weakest form of binding is *by type*: the EU requires that, at any moment, the bound resource is compliant with a given type, no matter what its actual value or identity are. This kind of binding is exploited typically to bind resources that are available on every CE, like system variables, libraries, or network devices.

Note that it is possible to have different types of binding to the *same* resource.

The different data space management mechanisms[1] are illustrated on Figure 2.2. The figure shows, for each mechanism, the configuration of bindings before and after migration of the grayed EU. Let us consider a migrating executing unit U whose data space contains a binding B to a resource R:

- in a *by move* mechanism, the resource R is transferred along with the execution unit U to the destination CE and the binding is not modified (see Figure 2.2(a)). There are two alternatives: either the other bindings to R are removed (top of the figure) or they are converted to a network reference (bottom of the figure). This mechanism can only be exploited if R is a free transferable resource.

- in a *network reference* (or remote reference) mechanism, the resource R is not transferred and once U has reached its target CE, B is modified to reference R in the source CE. Every subsequent attempt of U to access R through B will involve some communication with the source CE (see Figure 2.2(b)). This mechanism can be exploited whatever the type of the resource and of the binding.

- in a *by copy* mechanism, a copy R' of R is create, the binding to R is modified to refer to R', and then R' is transferred to the destination CE along with U (see Figure 2.2(c)). This mechanism is suitable a priori in any case where the binding type is not by identifier.

- in a *rebinding* mechanism, B is voided and re-established after migration of U to another resource R' on the target CE having the same type of R (see Figure 2.2(d)). Rebinding is suited to by-type bindings: it exploits the fact that the only requirements posed by the binding is the type of the resource, and avoids resource transfers or the creation of inter-CE bindings. This mechanism requires that, at the destination site, a resource of the same type of R exists.

Fuggetta et al. analyze further the relations between the nature of resources, of the bindings to these resources and the consequences on the possible data space management that are applicable. They do achieve some

[1] In the following of this thesis, we also refer to such mechanisms as *resource management policies*.

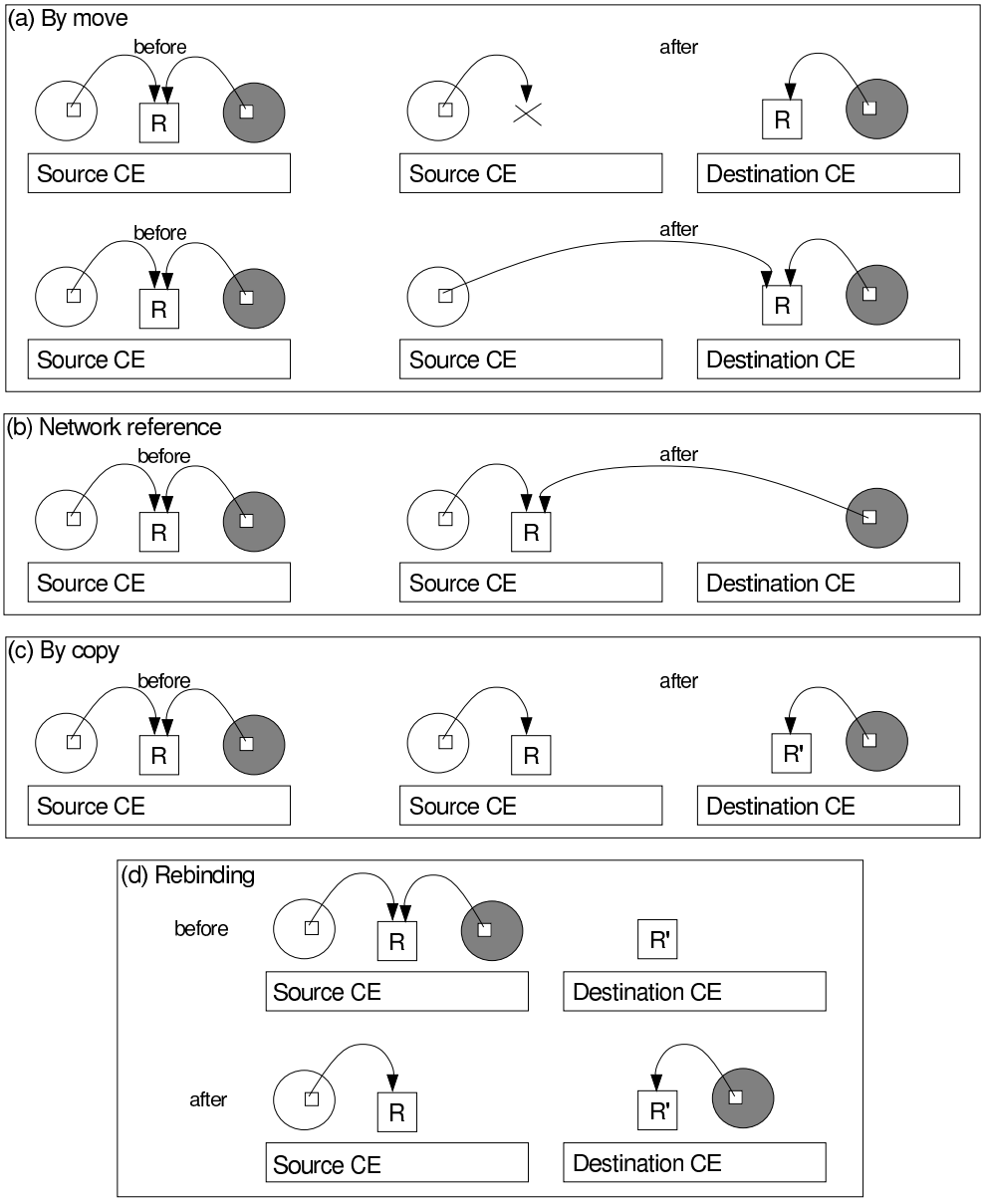

Figure 2.2: Data space management mechanisms.

kind of classification, but it is not absolute. In most of the cases, it eventually depends on application specific criteria. For instance, a *fixed transferable* resource, which is bound *by value* to an EU can support either a *by copy* or a *by network reference* mechanism.

The existing MCSs exploit different policies as far as data space management is concerned. However, the nature of the resource and the type of binding is often determined by the language definition or implementation, rather than by the application programmer, thus constraining the mechanisms exploited. We will come back to this issue later in this chapter since it is precisely that point that motivated our thesis work.

2.1.2 Applications of Code Mobility

As of now, applications exploiting code mobility are still very few in comparison to client-server based applications. This is a consequence of the immaturity of the technology on the one hand (in particular concerning performance and security [64]), and of the lack of appropriate methodologies for application development.

However, mobile code is expected to bring benefits by enabling new ways of building distributed applications and even creating brand new applications. This is very appealing in some application domains in particular.

The key benefits of mobile code are listed below:

Service customization. In conventional distributed systems built following the client-server paradigm, servers provide a fixed set of services through a statically defined interface. This set of services does not necessarily meets the needs of every user, thus entailing the server to be upgraded, increasing both its size and complexity without increasing its flexibility. Using code mobility, the server can provide a set of very simple and low-level services that are then composed by the client code to obtain the adequate high-level functionality.

Deployment and maintenance. Mobile code is proving useful in supporting these last phases of the software development process. Indeed, in a distributed setting, installing of rebuilding an application at each site still has to be performed locally and with human intervention. Conversely, a mobile program could visit each host and perform the operation automatically, or even the application itself could request a central repository for automatic update.

Autonomy. Mobile concepts and technologies embody a notion of autonomy of application components that is useful for coping with the heterogeneous nature of communication links (reliability, bandwidth). Using a mobile program, a set of several interactions with a server can be packed into one entity that needs to be passed only once through the

network, operate autonomously and independently, and finally transmit the final results to the node that sent it. This way, interactions between the sending node and the server are reduced to the minimum.

Fault tolerance. The action of *migrating* code and possibly sending back the results is not immune from the partial failures that can occur in any distributed systems. However, the action of *executing* code that embodies a set of interactions that should otherwise take place over the network is actually immune from partial failure.

Hereafter is a non-exhaustive list of the application domains that are expected to benefit from code mobility:

Distributed information retrieval. Such applications gather information matching some specified criteria from a set of information sources dispersed in the network. This application domain is very wide, encompassing very diverse applications. Code mobility could improve efficiency by migrating the code that performs the search process close to the information base to be analyzed [40].

Active documents. In active documents applications, passive data like e-mail or web pages are enhanced with the capability of executing programs which are somewhat related with the document contents. Code mobility is fundamental since it enables the embedding of code and state into documents and supports the execution of the dynamic contents during document fruition.

Advanced telecommunication services. Support, management, and accounting of advanced telecommunication services like video conference, video on demand, or telemeeting, require a specialized "middleware" providing mechanisms for dynamic reconfiguration and user customization—benefits provided by code mobility.

Remote device control and configuration. Such applications are aimed at configuring a network of devices and monitoring their states. This domain encompasses several other application domains, *e.g.* industrial process control and network management. In the classical approach, based on the client-server paradigm, monitoring is achieved by polling periodically the resource state and configuration by using a predefined set of services. This approach can lead to a number of problems [68]. Code mobility could be used to build monitoring components that are co-located with the devices being monitored and report events representing the evolution of the device state. In addition the shipment of management components to remote sites could improve both performance and flexibility [3].

Workflow management and cooperation. Workflow management applications support the cooperation of persons and tools involved in an engineering or business process. The workflow defines which activities must be carried out to accomplish a given task as well as how, where, and when these activities involve each party. A way to model this using code mobility is to represent activities as autonomous entities that, during their evolution, are circulated among the entities involved in the workflow [14].

Electronic commerce. Such applications enable users to perform business transactions through the network. The application environment is composed of several independent and possibly competing business entities. A transaction may involve negotiation with remote entities and may require access to information that is continuously evolving, *e.g.* stock exchange information. There is the need to customize the behavior of the parties involved in order to match a particular negotiation protocol. Moreover, it is desirable to move application components close to the information relevant to the transaction, for security reasons for instance. All these problems make mobile code appealing for electronic commerce applications. Actually, Telescript [66] was conceived explicitly to support electronic commerce.

2.1.3 Code Mobility in Java

Java has triggered most of the attention and expectations on code mobility. Most of the research work done in the field today is performed using Java. Apart from systems developed in Java, the Java language itself supports code mobility.

The Java compiler translates Java source programs into an intermediate, platform-independent language called *Java Byte Code*. The bytecode is interpreted by the *Java Virtual Machine* (JVM)—the CE implementation. Java provides a programmable mechanism, the *class loader*, to retrieve and link dynamically classes in a running JVM. The class loader is invoked by the JVM run-time when the code currently in execution contains an unresolved class name. The class loader actually retrieves the corresponding class, possibly from a remote host, and then loads the class in the JVM. At this point, the corresponding code is executed. In addition, class downloading and linking may be triggered explicitly by the application, independent of the need to execute the class code. Therefore Java supports weak mobility using mechanisms for fetching code fragments.

Several mobile agents systems have been developed in Java, *e.g.* Aglets [42, 44], Mole [55], Ajanta [63]. All these systems support weak mobility, and adopt a fix mechanism for data space management. Aglets uses a *by copy* mechanism, whereas Mole uses a *by move* mechanism. Anyhow, the choice

is fixed.

Java by itself does not support strong mobility. However, there are several attempts to achieve it, based on different approaches [8]. The problem is that such approaches either require a modification of the JVM and/or its interpreter ([7]), therefore sacrifying portability, or they use some preprocessing ([33, 53]), which is very costly. The problem comes from the fact that Java programs do not naturally have access to the internal state information of threads.

2.2 Meta-Programming and Reflection

Reflection in programming languages dates back to the seminal work of Smith in the early eighties [54]. It was introduced in object-oriented programming by the famous work of Pattie Maes [47]. It is basically the ability of a system to watch its computation and possibly change the way it is performed.

2.2.1 Principles

Bobrow et al. noticed that there are two aspects of reflection, observation and modification:

> "Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: *introspection* and *intercession*.
> Introspection is the ability for a program to observe and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state or alter its own interpretation and meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called *reification*." [6]

An object-oriented reflective system is logically structured in two (or more) levels, constituting a *reflective tower*. The first level is the *base-level* and describes the computation that the system is supposed to do. The second one is the *meta-level* and describes how to perform the previous computations. The entities working in the base level are called base-entities, while the entities working in the other level(s) are called meta-entities. *Meta-programming* is the activity of programming meta-level entities.

Each reflective computation can be separated in two logical aspects: computational flow context switching and meta-behavior [17]. A computation starts with the computational flow in the base-level; when the base-entity begins an action, such an action is trapped by the meta-entity and the

computational flow raises at meta-level (*shift-up* action). Then the meta-entity completes its meta-computation, and when it allows the base-entity to perform the action, the computational flow goes back to the base level (*shift-down* action).

There are two kinds of reflection, *behavioral reflection* and *structural reflection*. The former can be defined as the ability to intercept an operation such as method invocation and alter the behavior of that operation. The latter is the ability to *alter* data structures used in a program, which are statically fixed at compile time. Some kind of language extensions require structural reflection for implementation, and thus cannot be simply implemented using behavioral reflection.

Reflective systems differ in the type of reflection they provide, as well as in the nature of the meta-entities. There are four recognized reflective models in this regard [17]:

the meta-class model (MCM). In this model, the reflective tower is realized by the instantiation link [24, 13]. The meta-object reifying a base-entity is its class, the meta-object reifying a meta-object is its meta-class, and so on. This main problem of this model is the difficulty of specializing the meta-behavior for a single instance, since any instance of a class has the same meta-object. Also, dynamically changing the behavior of an object implies substituting its meta-class, which can lead to inconsistencies and is not offered by all languages.

the meta-object model (MOM). In this model, the reflective tower is realized by the clientship relation [38]: separate entities handle intercession and introspection on each base-entity. With this approach it is simple to specialize the meta-behavior per object. The major drawback of the model is that a meta-object does not have access to the sender's identity when monitoring a message. It is the most used model, with applications in several areas.

the message-reification Model (MRM). In this model, meta-entities are special objects called *messages* which reify the actions that should be performed by the base-entities [28]. Every method call is reified into an object which is charged with its own management and exists only for the duration of the action it embodies. The major drawback of this model is the lack of information continuity, since it is impossible to store information among meta-computations.

the channel reification model (CRM). This model is an extension of the message reification model [2]. It is aimed to overcome some of its limits, in particular that of information continuity, while keeping its advantages.

2.2.2 Applications of Reflection

Reflection allows deriving new behaviors from initial ones by introducing some variations of the computational model. Separation of concerns, reusability, extensibility and flexibility are some of the main advantages of reflection. Therefore, computational reflection is a programming paradigm suitable to develop *open systems* (*i.e.*, systems that can be extended in a simple manner).

Reflection has been applied to various application domains, such as compilers [41], operating systems [69], distributed systems [49]), middleware [45, 5], fault tolerant systems [26] and graphic interfaces [51].

In this thesis we apply it to mobile object systems, a special kind of distributed systems, to introduce flexibility in the way resources (passive objects) attached to migrating entities are managed.

2.2.3 Reflection in Java

Java is a programming language supporting reflection. The ability of Java in reflection is called the reflection API [61]. However, it is restricted to introspection: for instance, Java enables a program to know the names of the methods in a given class and to instantiate the class with a given string name. On the other hand, it does not enable to alter program behavior. Only can one modify the value of an instance variable, *e.g.*.

To address the limitations of the Java reflection API, several extensions have been proposed, most of them enabling behavioral reflection. The runtime systems of those extensions call a method on a *metaobject* for notifying that an operation is intercepted. The programmer can define their own version of the metaobject so that the metaobject executes the intercepted operation with customized semantics. This follows the meta-object model (MOM) exposed above.

Some kinds of language extensions, such as the Reflex system exposed in this thesis, require structural reflection. As of now, structural reflection for Java is offered only by the Javassist API [21], on which Reflex is based.

2.3 Reflection for Code Mobility in Java

Reflection can be used to achieve adaptability in various domains, and in particular to that of code mobility. In section 2.1.1 we have presented the two areas where code mobility mechanisms are involved: code and execution state management, which can be based either on strong mobility or on weak mobility, and data space management, which can rely on mechanisms such as by copy, by network reference, rebinding, etc. We have highlighted that in present systems, the mechanisms used are fixed choices, though application requirements could imply the need for other mechanisms.

The point is that reflection could be used to make these choices adaptable to the programmer's needs. In particular, we think that fixed choices in data space management mechanisms are not suited to the dynamical and unpredictable nature of networks. Thus we aim in this thesis at developing a system that can allow for customized specification of the data space management mechanism (or resource management policy) to be used. We use reflection for that purpose, in Java. The remaining of this dissertation is the presentation of our work in this regard.

A possible application of easily specifiable policies for data space management is in the mobile agent domain [46]. Network traffic, network reliability, CPU performance, access to memory and resources topology are indeed subject to change, *i.e.* network characteristics are not defined statically. Therefore, if these resources are reified, then an agent could dynamically introspect them and adapt its resource management policy depending on them.

For instance, in the case of a free transferable resource, if we deal with a low-reliable network (*i.e.* a disconnection can occur), then the by-reference mechanism should be avoided; if we deal with a low-bandwidth network and moderate network traffic, then the by-move policy should be avoided.

Reflection could also be used to make code and execution state management policies dynamically interchangeable, as sketched out in [46]. The authors claim that the policy to be used (weak mobility, strong mobility, or even remote evaluation or code on demand) should not be fixedly chosen by the system. Instead, depending on the entities that do need to be migrated, a particular policy can be chosen.

The concrete realization of these ideas is still to be done, since as far as we know only concepts have been formulated on the topic. Reflex, the reflective system developed during this thesis, is a first concrete attempt to achieve such adaptability in code mobility in Java.

2.4 Summary

In this chapter, we have introduced the necessary concepts used in this dissertation.

We presented what code mobility is about and how to look at it in a structured way, analyzing its different characteristics. Limitations in flexibility of existing mobile code systems have been highlighted.

Then we exposed the core concepts of reflection and meta-programming, which can help in developing open and adaptable systems.

Finally, the technical motivation of this thesis has been put into light: using reflection to solve the limitations in flexibility of Java mobile code systems. This is a particular application of a reflective system we developed, called Reflex.

Chapter 3

Reflex

In order to achieve flexibility in the specific area of resource management in mobile object systems, we have designed and implemented a Java system called *Reflex* whose aim is to provide ***transparent reflective objects in Java***.

A *reflective object* is an object that has an associated meta-behavior, altering or extending its normal behavior. Similarly, we will refer to a *reflective class* as being a class whose instances are reflective objects. By *transparent*, we refer to the property of transparent type-compatibility between a reflective object and its non-reflective equivalent. This property implies that a reflective object can be assigned to a variable declared as a non-reflective one—as opposed to interface-based approaches, where the variable to assign the object to has to be declared as being of the type of the interface that the reflective object implements.

In this chapter, we present the Reflex system, its motivation, design and implementation, along with some examples of use, not restricted to our target application domain. As the Reflex implementation relies upon an existing library for reflection in Java, *Javassist* [20, 21], we also briefly explain the principles and functionalities of this library (in section 3.3.1).

3.1 Introduction

In general, having a *reflective* version of an object can be useful to implement specific (meta-) behaviors while preserving intact the base implementation of the object, achieving convenient separation of concerns. This use is unrestricted, offering a virtually infinite set of areas of application. This is after all what makes reflective techniques so fascinating and at the same time so dangerous: the best (the most meaningful) as well as the worst (the most absurd) can be done very easily.

In our case, we foresaw that we could fulfill our needs if we had an adequate system to get reflective objects. In fact, all the semantics of re-

source management policies in mobile object systems could, supposedly easily, be implemented as meta-behaviors in metaobjects. A resource requiring a particular management policy could be associated with a metaobject implementing the policy semantics. By locating the policy semantics in the resource itself (its metaobject) and not in the objects referencing the resource, we could ensure a consistent use of the resource. Also this solution seemed lighter than a centralized approach, with cooperating central managers in each host ensuring the correct semantics for all special resources located within themselves. This assumption was still an assumption when we undertook the development of Reflex—we will validate it further in this thesis report.

There are two kinds of existing systems that achieve similar behavioral extensions of objects that the one we aim at, but neither of them did meet our requirements:

- The first kind rely upon the use of Java interfaces to achieve type-compatibility ([15, 60]). They lack what we call *transparent type-compatibility*. In section 3.2.1 we argue why we consider this a major issue.
- The second kind, namely libraries for behavioral reflection in Java ([65, 39, 35, 67]), provide transparent type-compatibility, but are not sufficient since they have the drawback of modifying the original class in some way (modifying the class source file or its bytecode representation). We expose this issue in more details in section 3.3.1, when discussing our choice of the Javassist library as a technology provider for our system.

Since we could not find a satisfying system, we undertook the development of our own, baptized Reflex, with the following major requirements:

- a reflective object should be *transparently* type-compatible with its non-reflective equivalent;
- the system should be able to create reflective objects or convert existing objects to reflective ones;
- dynamically controlling the layer of meta-behaviors associated with a reflective object should be possible, easy, and consistent;
- it should be sufficiently open to be adaptable to unforeseen uses.

3.2 Motivation and Objectives

In this section we discuss the motivation of each of the four objectives stated above, as well as their achievement.

3.2.1 Transparent Type-Compatibility

Types and type-compatibility in Java

Java [56] is a *strongly typed* language, which means that every variable and every expression has a type that is known at compile time. Types limit the values that a variable can hold or that an expression can produce, limit the operations supported on those values, and determine the meaning of operations.

The types of the Java language are divided into two categories: primitives types and reference types. Primitives types are the boolean type and the numeric types. Reference types are class types, interface types, and array types. There is also a special null type.

A variable of a primitive type always holds a value of that exact type. A variable of a class type T can hold a null reference or a reference to an instance of class T or of any class that is a subclass of T. A variable of an interface type can hold a null reference or a reference to any instance of any class that implements the interface.

As stated in the Java language specifications [56], a reference to an object of type S (source) is assignable to a variable of type T (target) if:

- if S is a class type:
  - if T is a class type, then S must either be the same class as T, or S must be a subclass of T;
  - if T is an interface type, then S must implement interface T.
- if S is an interface type:
  - if T is a class type, then T must be `Object`;
  - if T is an interface type, then T must be either the same interface as S or a superinterface of S.

In other words, type-compatibility of two objects is equivalent to assignment-compatibility of their respective types (class type or interface type), and this is achieved either by type equality or by subtyping. In the Reflex system, type-compatibility between a reflective object and its non-reflective equivalent is achieved by subtyping, *i.e.* the type of the reflective object is a subtype of that of its non-reflective equivalent.

The need for transparent type-compatibility

An important factor of acceptance of new systems in general is their ability to integrate seamlessly into existing systems. Introducing a new aspect into a system should ideally not entail rewriting part of it.

Part of the existing systems that generate specialized versions of objects achieve type-compatibility through the use of Java interfaces (such as

the ProActive library for distributed computing [15], or the Dynamic Proxy Classes of the upcoming JDK1.3 [60]). The class of the object to convert is assumed to implement a set of interfaces that specify all or part of its services, and the generated object is typically an instance of a class which implements the same set of interfaces, and implements the specialized behavior.

This approach works fine only if the host system was entirely developed with interfaces in mind, or if the whole system is being developed from scratch. Moreover, it implies a constraint on the kind of objects that can be specialized. Indeed, we might want to specialize objects that are reused from an existing library—the Java class library being the most obvious case. In such case, a system that relies on interface-based type-compatibility can be problematic, as illustrated hereafter.

Let us consider an existing mobile-agent application, where a particular mobile agent has a method that accepts as parameter an instance of the class `java.util.Vector`. Now let us consider that we want to specialize the behavior of a particular vector, say, using a reflective object with a log feature metaobject that logs every operation performed on the vector into a file. As defined in the Java class library, the `java.util.Vector` class implements the `java.util.List` interface, which defines some, *not all*, of the operations that can be performed on a vector. Using an interface-based system, one can only generate an object whose type is compatible with the `List` type (*i.e.* it is an instance of a class that implements `List`). There are two important limitations to this:

1. Obviously, it is impossible to invoke on the generated object methods which are specific to the class `Vector`, not contained in `List` (*e.g.* `setSize()`, `firstElement()`, `capacity()`, etc.). To solve this issue, one can only create an exhaustive interface for a vector, build a subclass of `Vector` and make it implement that interface.

2. Worse, if ever the mobile agent class was developed using `Vector` as the argument type for the method we are considering (which is obviously the most frequent case), it is needed to modify the code and replace `Vector` with the name of the interface to use. Moreover there are cases where the source code of the client accessing the vector is not even available...

This simple example illustrates what we consider a major drawback of the *practical use* of interface-based programming. Restricting ourselves to systems like Reflex that generate reflective objects, we claim that it is crucial for the integratability of such systems (and consequently their acceptance) that the generated objects *are* of the same type than the source object. In our example, if the reflective objects system can create a reflective vector

which is an instance of class `java.util.Vector`, then no rewriting is necessary: the mobile-agent *does not even know* that the vector it is receiving is different from a standard one. This is true *transparency*.

For these reasons, Reflex generates reflective objects that are instances of a *reflective* subclass of the class we want to instantiate. The motivation for building reflective subclasses is exposed in section 3.3.1, and the process itself is explained in details in section 3.3.2.

3.2.2 Creation of Reflective Objects

Conceptually, a reflective object is the composition of a base object and a meta-behavior (one or more metaobjects). The combination between both is done when the reflective object is instantiated. The creation of a reflective object falls into two distinct cases, depending on whether the basic object already exists or not.

Basic Instantiation

In the most simple case, the object itself is not yet created, thus we want a convenient instantiation mechanism that allows us to specify the class from which the object must be instantiated and the metaobject to be associated with it.

The `Reflex` class, the main class of our system, has a simple method to instantiate a reflective object:

```
A o = (A) Reflex.createObject(``mypackage.A'',
                              new MyMetaobject());
```

This method returns an object which *is* of type `A`, and whose associated metaobject is an instance of `MyMetaobject` (an example class of metaobjects). The returned object is in fact an instance of a *reflective* subclass of class A, which has been generated automatically (as explained in section 3.3.2).
Note that there exist overloaded versions of this method for specifying arguments that should be passed to the constructor of the object. Specifying the metaobject at creation-time is not compulsory, it can also be specified later.

Conversion

The creation scheme presented above is sufficient most of the time, when the programmer has easy access to the line of code where the object is instantiated, but might be problematic in other cases.
For example, a design pattern that is frequently used in object-oriented programs is the *Factory* pattern [34]. When using code developed according

to such a creational design pattern, the line where the concrete instantiation of the object occurs is generally not accessible to the programmer. Obviously in such a case we need a way to convert an already-created object to a reflective object.

For this purpose, the Reflex class has a converting method:

```
// get the object from a factory
A o = MyFactory.getInstance();

// convert it
o = (A) Reflex.convertObject(o, new MyMetaobject());
```

This method returns a new object, which is a reflective *shallow clone* of the given object. Recall that shallow clone means that the value of the fields of a shallow-cloned object are not cloned themselves. The returned object has exactly the same fields as the original one, and each field value is either a copy of the original field value (if it is a primitive type) or a reference to the actual value of the original field (if it is an object).

Obviously, as such a conversion does not actually change the object, but instead returns an enhanced equivalent, it must be performed before any references to the object are given to the outside world. Otherwise, some objects in the application will still hold references to the "unconverted" object, while others will have references to the reflective one.

3.2.3 A Framework for Metaobjects Composition

Obviously, developing metaobjects is similar to developing any other kinds of objects. After all, a metaobject is just a normal object that has meta-level properties. Thus, all the concerns of object-oriented development are still applicable, in particular *reusability*. All the more that meta-level programming is generally acknowledged as a difficult and potentially dangerous task (because of its system-wide implications).

Since meta-level programming is achieved using the same programming paradigm than that of the base level, namely object-oriented programming (hereafter OOP), reuse of metaobjects should be obtained by taking advantage of the two traditional ways to compose behaviors in OOP, namely aggregation and specialization.

Therefore, in the light of previous work that has been done on *explicit composition of metaobjects* [50], we consider that a consistent composition mechanisms for metaobjects is crucial in a system like Reflex. This composition mechanism should not only address the issue of composing orthogonal customizations but as well those having potentially overlapping semantics.

The composition mechanism we have adopted for Reflex relies on the concept of *cooperative metaobject* introduced in [50]. A cooperative metaobject can be considered as a metaobject containing a "hole" into which

may be inserted another metaobject. That metaobject can itself be cooperative, or non-cooperative, thus making a *chain of metaobjects*. A cooperative metaobject is designed keeping in mind that it needs to collaborate with the metaobject it is aggregating, if any. As any other metaobjects, it strictly respects the contract enforced by the metaobject protocol [38] (hereafter MOP) and should not make any assumptions on its aggregated metaobject other than the respect of this contract. Therefore, for each request it is its responsibility to perform possible pre-processing, forward the request to its aggregated metaobject, and then finally perform possible post-processing.

As a matter of fact, there is an unbreakable limit to composition of metaobjects: there can be only one value returned to the base-level. It is then up to the designer of metaobjects to ensure the consistent semantics. Generally speaking, a non-cooperative metaobject (at the end of the composition chain) will be called a *semantically strong* metaobject (*e.g.* implementing a new message-passing semantics), whereas a cooperative metaobject is, in our terminology, either an *auxiliary* behavior provider (*e.g.* implementing a tracing feature) or an *adaptor* to a semantically strong metaobject. In the case of an auxiliary metaobject, the returned value is the one returned by the semantically strong metaobject, whereas in the case of an adaptor, the returned value might be changed[1]. As of now, composition details are left as the programmer's task, since we could not find a generic solution to ensure composition consistency (see section 6.3.3).

The composition mechanism of Reflex allows for *dynamic composition and decomposition* of metaobjects. It offers methods for adding a metaobject (which must, obviously, be a cooperative one[2]) and removing a metaobject from the chain:

```
A o = (A) Reflex.createObject(``mypackage.A'',
                              new MyMetaobject());
... // do something here until you need,
    // say, a log metaobject

LogMetaobj log = new LogMetaobj(``logfile.out'');

// add it to the chain of metaobjects
Reflex.addMetaobject(o, log);

... // do logged actions

// remove it
```

[1]This terminology of metaobjects is ours, and simply aims at giving a general feeling on the role of each kind of metaobject. It is still in a very immature state, therefore this part of the discussion should not be seen as an immutable rule.

[2]Exception made of the case where the metaobject to add is the first one in the chain.

```
Reflex.removeMetaobject(log);

... // go on
```

There is however a consistency concern with being able to remove metaobjects at any time. For instance, a metaobject could have set up an infrastructure to achieve its role, and removing it at that time could leave the system in an inconsistent state. Or it could be simply meaningless to remove such a metaobject. This is why the result of the `removeMetaobject()` call is not guaranteed to work: the metaobject can *refuse* to be removed.

We will introduce other particularities of our framework when presenting the meta-level architecture in more details (section 3.3.3).

3.2.4 An extensible MOP

When designing the MOP of Reflex, we considered important to accept the fact that we could not make it sufficiently complete for all kinds of needs. Thus the solution we adopted is to let an "open-door" in the MOP that can be used to freely extend it without having to modify it.

To achieve this, we have introduced the concept of a *metamessage*. Mainly, a metamessage is a message that can be sent from the base program to the layer of meta-behaviors of a reflective object. A meta-behavior implementation (a metaobject) is free to process or not any incoming metamessage. Such a metamessage is characterized by a name, which identifies its *kind*, and by an array of objects being the arguments to that message:

```
A o = (A) Reflex.createObject(``mypackage.A'',
                              new MyMetaobject());

// pack arguments for the metamessage
Object[] args = {``on''};

// send it
Reflex.sendMetamessage(o, ``setDebugMode'', args);
```

The MOP enforces that a metaobject will accept incoming metamessages. However, it makes the assumption that a properly-acting metaobject will forward the metamessage to the other members of the meta-layer, may it process it or not.

Though the primary aim of a metamessage is to open a free communication channel between the base level and the meta level (*inter-level communication*), its use can be extended to communication between metaobjects themselves (*intra-level communication*). In such a case, the assumption that a metaobject systematically forwards each metamessage to the other metaobjects in the chain does not hold anymore. It is left up to the designer to address this issue.

The metaobjects we have developed during this thesis make use of this feature, as we will explain in a next chapter. A particular kind of metaobjects, targeted to distributed environments, understand an extra MOP message, that we implemented as a metamessage. This metamessage is used to control the serialization process of a reflective object. Similarly, we are convinced that this feature will be helpful to people using Reflex and wanting to extend the present MOP.

3.3 Architecture of Reflex

The Reflex system is centered around the `reflex.Reflex` class, whose public interface offers methods to create a reflective object, convert an existing object to a reflective object, communicate with the layer of meta-behaviors of a reflective object, and add/remove metaobjects to this layer.

As we saw before, in Reflex a meta-behavior is implemented as a *metaobject*. Such a metaobject is a standard Java object that respects a given protocol, the MOP, and implements actions that should be taken in some circumstances. The *hooks* that activate the meta-behavior are principally the method calls performed on the base object[3]. Such a method call is first *reified—i.e.* made accessible in terms of the programming language itself. It is then passed to the metaobject which can analyze it and operate accordingly. To achieve this, we need a system that provides *behavioral reflection* in Java, that is to say a system that is able to somehow insert the hooks necessary to trigger the meta level processing.

We finally adopted the Javassist library for our needs. The motivation of this choice as well as a quick presentation of it is the subject of the following section.

3.3.1 The Javassist Library

Javassist [20, 21] is a class library enabling structural reflection in Java. Since portability is a crucial issue in Java, it relies on a new architecture for structural reflection that can be implemented without modifying an existing runtime system or compiler, since it operates at *load time*, *i.e.* when a class is loaded into a JVM. Javassist is a Java implementation of that architecture. An essential idea of this architecture is that structural reflection is made possible only before load time and it is achieved by equivalent bytecode transformation. After a program has been loaded into the JVM, it is not possible to perform structural reflection anymore.

[3]Meta-behavior can also be activated by sending metamessages, as explained in section 3.2.4.

Javassist and other extensions for reflection in Java

A great feature of Javassist is that it provides high-level abstractions, therefore entailing that users of this library do not need to have a deep understanding of the Java bytecode (conversely to other libraries for bytecode transformation such as the JavaClass API [25] and JOIE [23]).

The compile-time metaobject protocol [19] is another architecture enabling structural reflection without modifying an existing runtime system. OpenJava [62] is a Java implementation of this architecture. This architecture was mainly designed for off-line use at compile time, since it operates on source code files, whereas Javassist operates on bytecode files. There are two major advantages for Javassist as opposed to compile-time approaches:

- Javassist can operate on classes even if the source code is not available (*e.g.* when the class is provided by a third party);
- Javassist operates faster than compile-time based systems, as demonstrated in [21], since it does not require compiling source code.

Finally, since behavioral reflection can easily be implemented on top of Javassist (and *is* actually implemented and included in the distribution of Javassist), Javassist indirectly covers applications of behavioral reflection systems for Java such as Kava [65], MetaXa [39, 35] and Reflective Java [67].

The need of structural reflection for Reflex

As a matter of fact, behavioral reflection could be sufficient for what we want to achieve with Reflex, that is to say creating reflective objects in order to alter behavior through metaobjects implementations. Thus we could have used a system restricted to behavioral reflection like Kava, Reflective Java, MetaXa. Hooks to perform the interception of method invocations is implemented by performing source-to-source translation before compilation in Kava, and by performing bytecode-level transformations just before a class is loaded into the JVM in Reflective Java and MetaXa.

In these systems, transparent type compatibility (3.2.1) is achieved by directly modifying the class of which a reflective instance is needed (at the source level or at the bytecode level). Thus reflective objects are obviously of the same type than normal objects, since it is the type itself that is updated. The problem with this approach is that *all* the instances of a given class are reflective. This might not be what is required, and implies an unnecessary cost. For instance, if in an application one needs *one* reflective vector, then, since class `java.util.Vector` will be updated, all the vectors used in the system will be reflective too. In addition to the performance cost, in a distributed environment, there could be class compatibility issues if a running host already loaded the "normal" version of that class[4].

[4]If a remote host has already loaded class Vector, it will not reload it (for instance

We did not want to include those drawbacks in Reflex. This is why we adopted the idea of creating a subclass, which is reflective, for any class that we want (at least) one reflective instance of. Thus a reflective vector is concretely an instance of a class generated by Reflex, which is made reflective. The original class `java.util.Vector` is *not* modified at all. The process of creating this class is explained in details in section 3.3.2, along with the class structure of the framework we developed to make this process specializable.

Thus, we need a system that provides structural reflection in Java in order to create these reflective subclasses dynamically. Until Javassist appeared, no systems addressed this issue of structural reflection in Java, though it could be indirectly achieved using low-level bytecode transformers such as JOIE and the JavaClass API.

To sum up, we need both types of reflection: structural to dynamically create the reflective classes, and behavioral to have our runtime adaptations running. Javassist is the only library available at present that offers both and, moreover, in a very elegant and high-level way.

Structural reflection with Javassist

Javassist provides several methods for creating a new class dynamically, as well as methods for altering class definitions. These methods can be divided into three main categories:

- Methods for changing class modifiers: making the class public, abstract, or removing the final modifier from the class;
- Methods for changing the class hierarchy: changing the class name, changing the superclass, changing the interfaces implemented by the class;
- Methods for adding a new member: adding a constructor, adding a (possibly abstract) method, adding a wrapper to a method, adding a field. It is also possible to alter a method body, by specifying the body of another (compiled) method to copy.

All those features are made available through a very high-level API, designed in such a way that it is difficult to wrongly produce a class rejected by the bytecode verifier of the JVM.

Behavioral reflection with Javassist

As mentioned before, behavioral reflection can be implemented on top of Javassist. The implementation of behavioral reflection relies on the structural abilities of Javassist. To make a given class reflective, Javassist modifies the bytecode of this class in the following way:

when a parameter value to some method is a Vector). If the loaded version is the normal version, then instances of it will not be reflective.

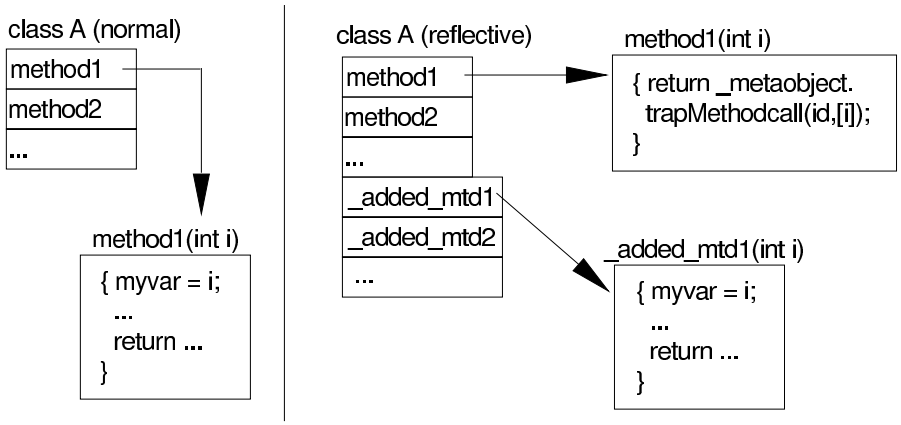

Figure 3.1: How Javassist makes a class reflective.

1. it makes the class implement the `javassist.reflect.Metalevel` interface, which defines methods to get and set the metaobject associated with an instance of the reflective class;

2. it adds to that class a field to hold a reference to the metaobject instance (the class from which this metaobject is instantiated is specified when making a class reflective);

3. for each method of that class, it renames it following an internal naming convention, and adds a new method, which has the original name, that just invokes the `trapMethodcall()` method of the metaobject (with parameters specifying the method being invoked and the arguments to that method), as illustrated on figure 3.1.

A Metaobject is created for every object at the base level. A different reflective object is associated with a different metaobject. The reflective version can directly be loaded into the JVM without updating the original class file, but it can as well be compiled to disk.

Finally, Javassist includes the `javassist.reflect.Metaobject` class, which serves as a base class for implementing classes of metaobjects. Among others, this class implements the methods listed in table 3.1.

More on Javassist

Our presentation of Javassist largely remains basic and limited to the features we make use of in our system. In particular, Javassist reflectional abilities normally include the trapping of accesses (read/write) to instance variables of a reflective class. However, this feature requires that a reflective class is known when the JVM is first started, so that the Javassist class

method name	description
`String getMethodName (int identifier)`	returns the name of the method specified by `identifier` (for efficiency reasons, methods of the base objects are indexed in a array, and thus are manipulated by identifiers, which represent their index in this array).
`Object getObject()`	obtains the object controlled by this metaobject.
`Class[] getParameterTypes (int identifier)`	returns an array of `Class` objects representing the formal parameter types of the method specified by `identifier`.
`Class getReturnType (int identifier)`	returns a `Class` object representing the return type of the method specified by `identifier`.
`Object trapMethodcall (int identifier, Object[] args)`	invoked when base-level method invocation is intercepted. The implementation of this method in the `Metaobject` class simply invokes the original method on the base object. It is normally overriden by subclasses. A subclass will typically refer to this method when wanting to invoke the base-level object method.

Table 3.1: Main methods of the Metaobject class of Javassist.

loader can modify the bytecode (*i.e.* insert hooks) of any class accessing instance variables of the reflective class. Obviously, this feature is not suitable in a distributed environment, moreover with the dynamic generation of reflective classes that we use.

For a complete description of the architecture implemented by Javassist as well as an exhaustive presentation of the API, we refer the interested reader to the original paper [21] as well as the Javassist' homepage [18] where all related documents as well as the library package can be found.

3.3.2 Building Reflective Classes

In this section we present the class building process to obtain reflective classes. But first of all, let us localize more precisely at which point of time comes the issue of building a reflective class.

Looking for the reflective class

Let `A` be the name of the class we want to have reflective objects of. As introduced before, we are first going to create a subclass of `A`, say `RA`, which will be the one instantiated to obtain reflective objects of type `A`.

When the following line of code is executed:

```
A o = (A) Reflex.createObject(``mypackage.A'',
                                  new MyMetaobject());
```

the `Reflex` class first checks if class `RA` is already defined and loaded into the JVM:

- If it is the case, then `RA` is instantiated, the new object is returned, and the process terminates.
- If `RA` is not yet loaded, but a file `RA.class` is available on disk, then `RA` is loaded into the JVM, instantiated, the new object is returned, and the process terminates.
- If `RA` does not exist at all, then it needs to be created. Here starts the class building process. Once created, `RA` is writtent to disk, loaded into the JVM, instantiated, the new object is returned, and the process terminates.

We have implemented Reflex so that reflective classes are built only if necessary, in a lazy-manner—only when we need to *instantiate* a given reflective class.

Class builders

To allow for customization of the class building process[5], we decoupled the responsibility of checking if the reflective class exists or not, possibly order its creation, and then instantiate it, from the responsibility of creating the reflective class. The first set of responsibilities is implemented by the `Reflex` class, whereas the effective class building process is implemented by what we call a *class builder*.

A class builder is responsible for allocating a name to the subclass it is creating, respecting a defined naming convention. From a class builder point of view, any class has only one reflective subclass. The name the class builder allocates to that subclass should be unique, and always the same (there is an equivalence relationship between a class name and the name of its reflective subclass, from the perspective of a particular class builder). Obviously, to avoid conflicts, two class builders should not have exactly the same naming convention.

To be used by Reflex, a class builder class should implement the interface `reflex.builder.ReflexClassBuilder`. This interface declares a method to get the name of the reflective subclass based on the name of a class, a method to order the creation of the reflective subclass of a class, and a method that defines the order relationship on class builders [6].

Class `reflex.builder.BasicClassBuilder` is a basic implementation of this interface. The naming convention of this class builder is extremely simple: the name of the new class is the concatenation of a package name for Reflex-generated classes (`reflexgen`) and the fully qualified name of the class we are creating a subclass of. For instance, the reflective subclass of class `mypackage.A` will be baptized `reflexgen.mypackage.A`.

A class builder is associated with a method factory, which is a class that implements the methods of the MOP that a reflective object should implement (those defined in the `ReflexObject` interface). For efficiency reasons, Javassist does not perform online compilation of source code, therefore methods need to be available in a compiled format in order to be inserted into a newly created class. A method factory class is just a repository of compiled methods that the class builder will use to insert into the class it is creating.

The process

The process of creating the reflective subclass is derived from how behavioral reflection is implemented by Javassist. We saw in section 3.3.1 that Javassist will trap all invocations of public methods and forward them to a specified metaobject. It does this by modifying all the public methods of the class. So,

[5] An illustration of such a customization is presented in chapter 4 when adapting Reflex to RM.

[6] This issue is examined in section 3.3.3 dealing with the integration of class builders within the metaobject composition framework.

in order to make *all* public methods trapped, we need to redefine (override) *all* the public methods of the class, in order to make them trapped later.

Thus, in the default version of the class building process, we want to create a class with the following properties:

1. it is a subclass of the specified class;
2. it implements the `reflex.ReflexObject` interface, since instances of it are reflective objects;
3. for each public method of the specified class (including those that are defined upper in the class hierarchy), there needs to be an overriding one (same name, same parameter types), that simply makes a super-call—in order to eventually use the true implementation of the method, which is found in a super class;
4. it has all the methods defined in the `ReflexObject` interface.

The structural reflectional abilities of Javassist cover all these needs. A class builder simply builds the reflective class sequentially, following these criteria one by one.

To make the super calls (step 3), the Javassist API contains a way to add a *delegator method* to a class. Such a method delegates its execution to a specified one (in our case, the one taken from the superclass that actually defines the method). To add all the methods of the MOP (step 4), Javassist provides a way to add a pre-compiled method to a class (the class builder gets it from its method factory).

When the class builder has created the class, it makes it persistent on disk so that next time it does need to be recreated, thus limiting the performance cost due to this process.

3.3.3 The Meta-level Architecture

As introduced in section 3.2.3, we have developed a framework for metaobjects composition within Reflex. In this section we present in details the architecture of the meta-level generated by Reflex, showing how the framework is implemented.

Chain of Metaobjects

Javassist has the limitation that it can only associate a metaobject to a base object in a static manner: the class of metaobjects to instantiate is specified once, when the reflective class is generated, and cannot be changed afterwards [7].

[7] This limitation was removed in version 0.7 of Javassist (most of our work was done with versions 0.5 and 0.6).

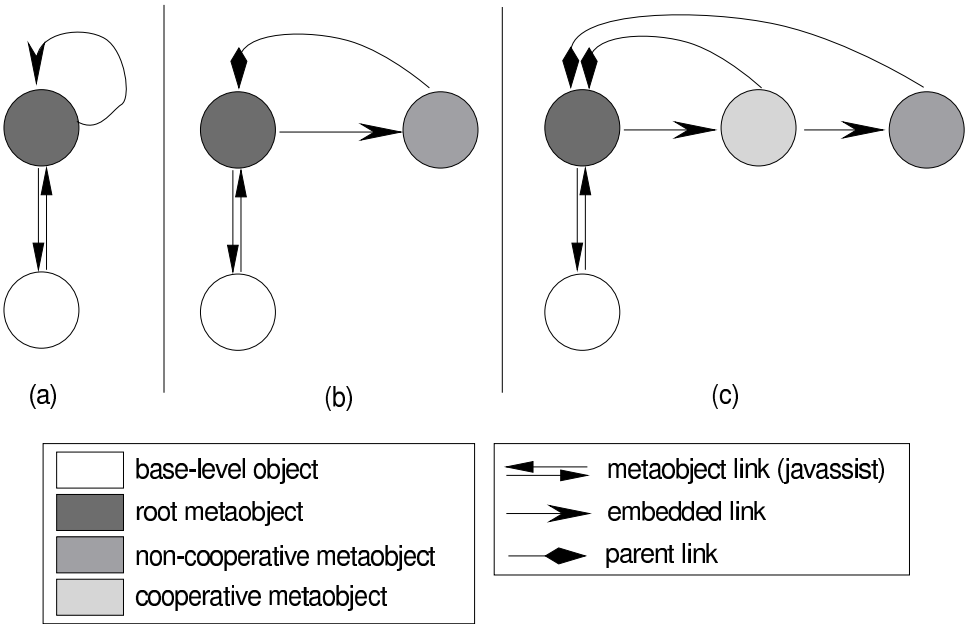

Figure 3.2: Metaobjects and their dependencies—(a) a root metaobject alone. (b) non-cooperative metaobject added. (c) cooperative metaobject added.

To allow for dynamic composition/decomposition of metaobjects, we have introduced a special kind of metaobject, a *root metaobject*. Such a metaobject is systematically associated with a reflective object and serves as the root of the chain of metaobjects. In itself, this metaobject does not modify the behavior of the base level object it is attached to. It just serves as a "hook" to hang other metaobjects. If a metaobject is attached to it, then it forwards requests to it (by default, it is self-attached). Moreover, an important role of the root metaobject is to manage the chain of metaobjects: it is its responsibility to add a metaobject to the chain, remove one, or remove all metaobjects of a given type[8].

Figure 3.2 illustrates the architecture of the meta-level in different cases. In 3.2(a), there is only the root metaobject in the meta-level layer. This happens if the reflective object was created without specifying any metaobject to associate with. The embedded link points to itself. In this configuration, every public method call on the base object is trapped by the root metaobject, which forwards it to itself, and just invokes the normal method on the base object.

In 3.2(b), a non-cooperative metaobject has been added to the meta-level

[8]This central role of a root metaobject remains sufficient to justify its presence, though it is not necessary anymore to dynamically change of metaobject since Javassist 0.7.

layer. The embedded link of the root metaobject now points to this new metaobject. Reciprocally, the non-cooperative metaobject has a parent link that points to the root metaobject. This link provides it with the services of a standard Javassist metaobject (see table 3.1).

Finally, in 3.2(c), an extra cooperative metaobject has been added. The root metaobject's embedded link now points to this latter metaobject. Like the non-cooperative metaobject, this metaobject has a parent link pointing to the root metaobject. It also has an embedded link (since it is aggregating the other metaobject) which is the collaboration link. Note that it has been added at the head of the chain, entailing that it will be the first to receive a request, and reciprocally the last to return any value.

Class Structure

The UML [31] diagram of the class structure of the metaobject composition framework is given in Figure 3.3. This framework has several specialization points:

- the interface `reflex.metaobj.ReflexMetaobject` can be implemented by any class claiming to be a class of metaobjects integratable with Reflex;
- the interface `reflex.metaobj.AggregatingMetaobject` can be implemented to create classes whose instances behave like cooperative metaobjects;
- for convenience, abstract class `reflex.metaobj.BasicMetaobject` can be extended to build metaobject classes;
- `reflex.metaobj.CooperativeMetaobject` is another convenient abstract class that can be extended to create classes of cooperative metaobjects.

The class `reflex.metaobj.RootMetaobject` implements the concept of root metaobject. It is a subclass of `reflex.metaobj.MethodMetaobject`, which is itself a subclass of `javassist.reflect.Metaobject`. The role of the `MethodMetaobject` class is just to withdraw the Javassist metaobject's initial capacity to trap accesses to instance variables[9]. Also, `RootMetaobject`, as subclass of `Metaobject`, offers all the services of this latter, *i.e.* getting the base object it is attached to, translating a method index into a method name, invoking the base object's method, etc. (see table 3.1).

A root metaobject aggregates a metaobject. It therefore implements the `AggregatingMetaobject` interface, which only defines accessors to the embedded metaobject. To be valid, a metaobject should be an instance

[9]In section 3.3.1 we explained why this feature cannot be used in our configuratio

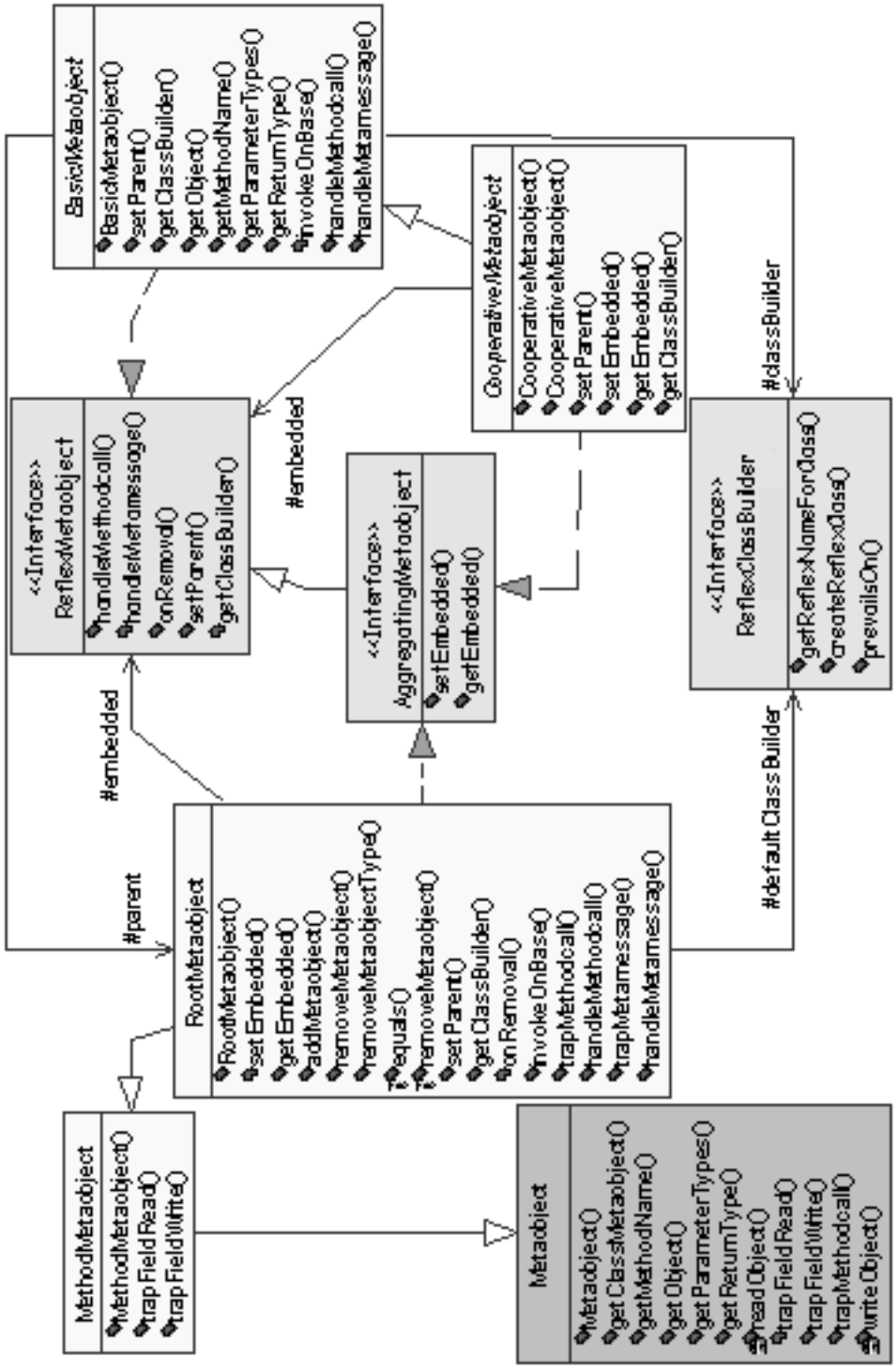

Figure 3.3: UML diagram of the metaobject composition framework

of a class that implements the `ReflexMetaobject` interface. This interface declares the protocol a metaobject should respect in order to be integratable with Reflex (presented in section 3.3.4). For convenience, we provide an abstract implementation of such a class of metaobjects, `BasicMetaobject`. This class implements all the services of a Javassist metaobject by forwarding requests to the root metaobject. It lets the implementation of the specific behaviors (*i.e.* handling method calls and metamessages) to its concrete subclasses. Our classes of metaobjects for resource management policies derives from this base class.

The framework includes another abstract class, `CooperativeMetaobject`, subclass of `BasicMetaobject` implementing the `AggregatingMetaobject` interface, which serves as a base class for cooperative metaobjects. The implementation of `CooperativeMetaobject` includes an appropriate implementation of the `getClassBuilder()` method (see later in this section). It also defines a special constructor taking as an argument an instance of `ReflexMetaobject`. This constructor can be used to specify the chain of composition directly when creating the reflective object. For instance, in the example below, we initially associate with the reflective object a chain of metaobjects consisting of a log feature metaobject, a tracing metaobject, and a semantically strong metaobject:

```
A o = (A) Reflex.createObject(``mypackage.A'',
      new LogMetaobj(new TraceMetaobj(new MyMetaobj()));
```

Metaobjects Composition and Class Builders

Since a class builder (see section 3.3.2) needs a method factory in order to insert all the methods of the MOP into the generated reflective class, there is a logical dependency between a particular MOP and the associated class builder.

Indeed, an extension of the MOP such as adding a method to it requires adding this method to the classes which are generated, that is to say modifying the method factory. Other extensions[10] might require adding other methods to the factory, or even new fields to the reflective class. Therefore, when the `Reflex` class determines that a reflective class must be built, it should use the adequate class builder, for the particular kind of metaobjects we want to associate (since metaobjects are *MOP-aware* entities, whereas neither `Reflex` nor the class of origin are). This dependency introduces some complications when considered along with the metaobjects composition framework.

Metaobjects are therefore aware of the class builder that should be used for them, since the latter embeds the MOP definition. The issue comes from the possible composition of metaobjects that do not require the same

[10] E.g., serialization awareness discussed in section 4.3.

class builders. This case occurs when reusing in an extended MOP some metaobject classes developed for a simpler version of the MOP.

The consistency issue can be solved by considering that MOP extensions are *compatible* with the base MOP, *i.e.* they add elements to the MOP without modifying the original ones[11]. Therefore, the class generated by a specialized class builder (extended MOP) is completely compatible with the class generated by the original class builder (normal MOP).

At the most abstract level, there must be a way to make a choice between class builders. For this purpose, we introduced an order relation on class builders that is defined by the predicate `prevailsOn(aClassBuilder)` (part of the `ReflexClassBuilder` interface). This predicate is true if the receiver should be used instead of the parameter one. The abstract class `CooperativeMetaobject` takes this predicate into account in order to respond to the `getClassBuilder()` message. In fact it returns the most prevailing class builder between its own, and the result of the invocation of the `getClassBuilder()` method on its embedded metaobject—thus ensuring the correct propagation of the most prevailing class builder through the composition chain.

We advocate that there is a hierarchy of class builders that reflects the specialization tree of the MOP. When composing metaobjects, the deepest class builder (the one that reflects the most specialized MOP) should be used. The default implementation of the `prevailsOn()` predicate is based on this principle: *a class builder prevails on another one if its class is a subclass of that of the parameter.* Obviously, another principle can be used, since the latter is just implemented in the `BasicClassBuilder` class.

There is however a limitation to this mechanism: to work properly, the metaobject that requires the most prevailing class builder should be specified at instantiation-time (since it is at that time that the class might be created). It cannot be added later when the reflective class has already been created. In practice, most probably the metaobject that relies upon the most specialized MOP is a *semantically strong* metaobject, specified at instantiation-time. However, we should not ignore other cases where, though we know at instantiation-time the class builder that should be used, we do not wish to associate such a metaobject instantly. For that purpose, the `Reflex` class offers overloaded versions of its `createObject` and `convertObject` methods that do accept as an extra final parameter the class builder that should be used to create the reflective class.

3.3.4 The Metaobject Protocol

In this section we summarize the metaobject protocol of Reflex. Although some of the methods mentioned here have been previously introduced in

[11] The base MOP was designed with this idea in mind: it should only embed the essential.

method name	description
`Object trapMethodcall (int identifier, Object[] args)`	invoked by the Javassist reflective core mechanism each time a public method is invoked on the baseobject. Arguments to this method are the identifier of the method and an array of objects that represents the arguments of the method being invoked. In the implementation of `RootMetaobject`, this method invokes the intra-level MOP method `handleMethodcall()` of the embedded metaobject.
`Object trapMetamessage (String message, Object[] args)`	invoked by the reflective object itself each time `sendMetamessage()` is called on it. Its arguments are the name of the metamessage and its arguments. In the implementation of `RootMetaobject`, this method invokes the intra-level MOP method `handleMetamessage()` of the embedded metaobject.

Table 3.2: Implicit part of the inter-level MOP.

scattered examples here and there, we considered important to give a clear exhaustive listing of the protocol.

The metaobject protocol can be split in two parts:

1. the part that addresses so called *inter-level communication*, that is to say the communication protocol between a reflective base object and its associated meta-layer;

2. the part that addresses *intra-level communication*, *i.e.* the communication protocol between metaobjects themselves.

Inter-level communication protocol

The first aspect of the inter-level communication protocol can be characterized as the *implicit* part of the protocol. It deals with the basic mechanisms, which are the method calls and the metamessages. The root metaobject has to provide two public methods for handling those cases. These methods, described in table 3.2 are *not* invoked by the base programmer explicitly.

The second aspect of the inter-level communication protocol is the methods that are made accessible to the base programmer in order to interact with the meta-layer associated with a given reflective object. Thus this part

method name	description
`Object sendMetamessage (String message, Object[] args)`	sends a metamessage to the chain of metaobjects. Its arguments are the name of the metamessage and its arguments (an array of objects).
`void addMetaobject (ReflexMetaobject rm)`	adds a metaobject to the chain of metaobjects. Its argument is a `ReflexMetaobject`, which in fact must be a cooperative metaobject if ever the chain of metaobjects is not empty.
`void removeMetaobject (ReflexMetaobject rm)`	attempts to remove the metaobject specified as parameter. The metaobject will be informed of this removal request through the invocation of its intra-level MOP method `onRemoval()`. It is free to refuse the removal operation.
`void removeMetaobjectType (Class type)`	attempts to remove all the metaobjects of the chain that are of the type specified as argument. Since this method relies upon`removeMetaobject()`, the same restrictions do apply.

Table 3.3: Explicit part of the inter-level MOP.

of the protocol can be characterized as *explicit*. These methods are defined in the `reflex.ReflexObject` interface, implemented by every reflective class generated by Reflex (see table 3.3).

Intra-level communication protocol

The protocol that a metaobject should respect is described in table 3.4, and defined in the `ReflexMetaobject` interface.

There are two other methods of this intra-level communication protocol that only concern cooperative metaobjects. They are defined in the `AggregatingMetaobject` interface, and implemented in our abstract implementation of a cooperative metaobject—`CooperativeMetaobject`. They are used by the root metaobject to manage changes in the chain of metaobjects (see table 3.5).

method name	description
`void setParent (RootMetaobject parent)`	called by the root metaobject in order to initialize the parent link of a metaobject. Recall that this link allows any implementor of `ReflexMetaobject` to have access to the services provided by a standard Javassist metaobject (see table 3.1).
`Object handleMethodcall (int identifier, Object[] args)`	This method is the core of the implementation of behavioral reflection in our system. Its implementation defines the specificity of a metaobject. For instance, a metaobject providing a log feature will do the log action in that method, before invoking the method on the base object. A cooperative metaobject will invoke this method on its embedded metaobject.
`Object handleMetamessage (String message, Object[] args, boolean treated)`	defines the handling behavior for metamessages. It is generally structured in a switch/case alike form, testing for the kind of the incoming message and eventually performing some actions. A cooperative metaobject will invoke this method on its embedded metaobject. The boolean parameter is used to inform whether the current message has been treated at least by one metaobject in the chain (if not, an exception will be raised).
`boolean onRemoval()`	called by the root metaobject when a removal request that concerns the metaobject has been received. Operations to perform before being removed will be implemented here. Note that this method returns a boolean indicating whether the metaobject accepts to be removed or not.
`ReflexClassBuilder getClassBuilder()`	initially called by the `Reflex` class in order to check for a reflective class and/or build one. Returns the class builder associated to the metaobject. Note that if the metaobject is a cooperative one, it will invoke this method on its embedded metaobject and return the most prevailing class builder of the two.

Table 3.4: Methods of the intra-level MOP.

method name	description
`void setEmbedded (ReflexMetaobject mo)`	sets the embedded metaobject of the receiver to the metaobject passed as parameter.
`ReflexMetaobject getEmbedded()`	returns the metaobject embedded by the receiver.

Table 3.5: Extra methods of the intra-level MOP for cooperative metaobjects.

3.3.5 The Reflex Public Interface

The public interface of the `Reflex` class is made up of two kinds of methods: the creational methods, that allow the programmer to get reflective objects, and convenient methods, which are just aliases to the methods defined in the `ReflexMetaobject` interface.

Creational methods

The creational methods have already been introduced in section 3.2.2.

- `createObject()`. This method returns a reflective object that is an instance of a reflective subclass of the class specified as parameter. Some metaobjects might be associated with it, if specified. This method is overloaded to accept different parameters to use for the constructor of the object. As explained in section 3.3.2, invoking this method might entail the whole process of creating a reflective class, if it does not exist yet.

- `convertObject()`. This method returns a reflective object that is a reflective shallow clone of the specified object. Some metaobjects might be associated with it, if specified. The shallow clone can be obtained in two different ways. If the class of the object to convert defines a constructor that takes as parameter an instance of this same class, then `convertObject()` uses this constructor. Otherwise, it makes use of the Java Reflection API [61] to initialize the fields of the new object.

Convenient methods

As a matter of fact, invoking a method of the inter-level MOP on a reflective object implies downcasting the object to `ReflexObject`. Since this is somewhat heavy, the `Reflex` class offers the same methods as those of the `ReflexObject` interface, taking as an extra first parameter the object on which to invoke the MOP method. The downcasting is then done by the `Reflex` class.

For instance, to add a metaobject, one would normally type the following code (`o` is a reflective object):

```
((ReflexObject) o).addMetaobject(new TraceMetaobj());
```

The same effect can be obtained by invoking the equivalent method on the `Reflex` class:

```
Reflex.addMetaobject(o, new TraceMetaobj());
```

All in all, this is just but syntactic sugar, though convenient one.

3.4 Summary

In this chapter, we have presented Reflex, a reflective system for Java.

We first highlighted the necessity of implementing a system of our own, since none of the existing systems achieved what we were aiming at. The requirements and main objectives of this system have been presented.

Then we detailed the architecture of Reflex, deeper into its design and implementation. The Javassist library on which Reflex relies has also been introduced.

The Reflex system we exposed here makes absolutely no assumption on the application domain. Though it is not adapted to operate fully in a distributed environment, it is open enough to make this extension smoothly.

Chapter 4

Reflex for Mobile Object Systems

The Reflex system we developed offers interesting perspectives to achieve flexibility in a variety of application domains. Within the context of our thesis, we have applied its use to the domain of resource management in mobile object systems.

In this chapter, we present how we extended Reflex so that it is suitable for a distributed environment. The idea in mind was to use it within a mobile object system, a particular case of distributed system with complex mobility. If Reflex is fully operational within a mobile object system, then it is also for other kind of distributed systems.

More precisely, we expose how we designed and implemented the interface with mobile object systems so that Reflex can be easily plugged in. Then we explain how we extended Reflex to make metaobjects "clever enough" to act in an distributed environment with migration.

This chapter still only focuses on the Reflex system, independently of any particular case. Examples of concrete use are the topic of the next chapter.

4.1 Introduction

The broad idea behind our adaptation, the one that in fact motivated us to develop Reflex, is that it is possible to obtain the different resource management semantics using metaobjects associated with these resources.

The concept is the following: when we determine that a given resource (which is nothing more than a Java object) needs to be dealt with in a particular manner, we make it reflective—either at instantiation-time or by converting it. Thus some metaobject is attached to it. The metaobject itself will then adopt the appropriate behavior to achieve the correct semantics it represents, depending on the *state* of its attached object.

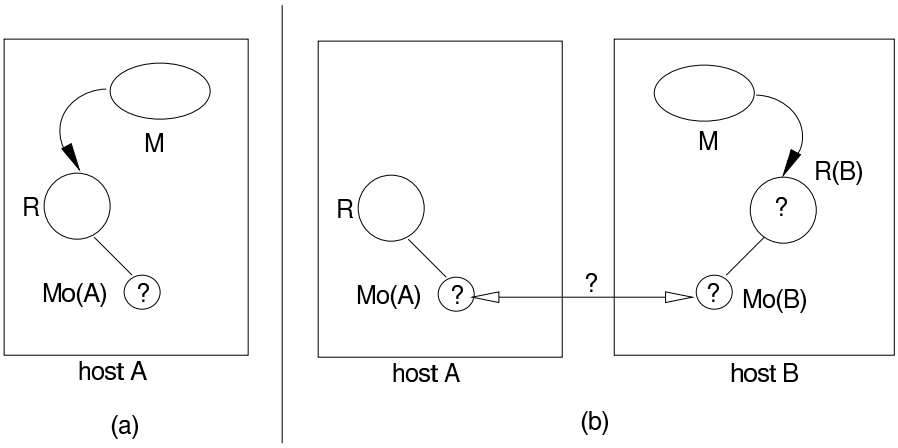

Figure 4.1: Reflective objects and migration—(a) A simple configuration before migration of the mobile agent. (b) After migration has taken place. The question marks indicate the specification points for implementing different semantics.

For instance, let us consider a resource R located on a site A, referenced by a mobile agent M. On site A, R has an associated metaobject that implements a given metabehavior Mo_A (Figure 4.1(a)). If M now moves to another host B, it will bring along with him a *copy* of R, say R_B, as well as a copy of its metaobject—since mobile object systems rely upon RMI [59], objects are passed by copy[1] over the network using the Java Object Serialization [58]. The metaobject should be aware of this migration and then adopt another metabehavior, Mo_B (Figure 4.1(b)). In fact, all resource management policies can be implemented by specifying what are exactly R_B, Mo_A and Mo_B, and how Mo_A and Mo_B interact if they do (the "?" on Figure 4.1(b)).

This principle implies that a metaobject should be able to know when migration is taking place, in order to adapt its own behavior (*i.e.* switch to Mo_B), and determine how its attached resource should be transmitted over the network (*i.e.* the content of R_B).

4.2 Interfacing with Mobile Object Systems

A major objective when designing the adaptation of Reflex for our application domain was its integrability with existing mobile object systems. Whatever mobile technology is used by the host system it should be quick and easy to integrate Reflex and the appropriate metaobjects to it. We even

[1]Except objects of the RMI remote type for which a stub to the object is passed, instead of the object itself.

aim at specifying an interface for such systems that do not require any single modification to an existing system. It is explained later on in this section.

4.2.1 Kinds of Serialization

In fact, as stated in the previous section, the only interface we need is to make metaobjects aware of *when* migration is actually taking place. Our solution to this issue comes from the observation of how systems providing object migration work: when migrating, an object (and all the objects it references[2]) is serialized and transmitted over the network. Most of the systems rely upon the standard RMI mechanisms for transmission over the network, though some systems (e.g., Aglets [42, 44]) use *extended* versions of the RMI streams.

In any case, migration cannot occur without serialization. Thus a metaobject needs to be aware of serialization of its base object—this is the topic of the next section. Once aware that serialization is happening, a metaobject should be able to determine the purpose of the serialization:

- Is it a serialization for *backup* (persistence)?
- Is it a serialization for *parameter passing* (remote method invocation)?
- Is it a serialization for *migration*?
- Is it for another purpose?

This distinction can be made by *looking at the concrete type of the stream used for serialization*—this is the idea of the interface between Reflex and mobile object systems.

4.2.2 Stream Identifiers

Therefore, we have introduced the notion of a *stream identifier*. Such an object is able to determine, given a serialization stream and a predefined type identifier, whether this stream is of that specified type. Since this knowledge is specific to the mobile object system into which Reflex is being integrated, it has to be explicitly specified at integration-time. For that purpose, a standard interface, `reflex.streamid.StreamIdentifier`, defines the method a class of stream identifiers should implement (see table 4.1) as well as type identifiers (static variables) that represent the three main types we have identified, namely backup, parameter passing, and migration.

The `reflex.streamid.DefaultStreamIdentifier` class provides a default implementation of that interface. Mobile object systems that rely upon the basic RMI streams can use it, in which case there is nothing at all to do to integrate Reflex. For instances of that class, if a stream is of type

[2]Exception made of RMI remote objects.

method name	description
`boolean identifyStream (OutputStream stream, int streamType)`	returns true if the given stream is of the specified type. Three types are defined in this interface, `BACKUP`, `PARAMETER`, and `MIGRATION`.

Table 4.1: The StreamIdentifier interface.

`sun.rmi.transport.ConnectionOutputStream`, then it is considered as a migration stream and a parameter stream. Otherwise it is a backup stream.

However, some systems might require a specialization of this default class. To allow automatic use of a specialized version, without having to modify anything in Reflex, the user stream identifier class should be called `reflex.streamid.UserStreamIdentifier`. This class must obviously implement the `StreamIdentifier` interface. Thus the interfacing with mobile object systems is reduced to the (optional) implementation of a class with a simple method.

Moreover, since the method from the `StreamIdentifier` interface takes as a second parameter an integer constant that represents the serialization type to check, it is possible for subclasses to define new constants if needed, in order to make a more specific classification of serialization types. This will not entail any modification of the actual framework.

4.2.3 The StreamMetaobject Class

In order to simplify the use of stream identifiers we have extended the Reflex framework of metaobjects with the `reflex.metaobj.StreamMetaobject` class.

This class is an abstract class that simply adds the stream identification abilities to its instances. It defines a new instance variable that holds a stream identifier object (of type `StreamIdentifier`), and implements the method of the `StreamIdentifier` interface by delegating to the stream identifier object. The correct initialization process for the stream identifier object is also implemented: it first looks if class `UserStreamIdentifier` is available, and if so, instantiates it; otherwise it instantiates the default stream identifier class[3]. Metaobjects that are instances of concrete subclasses of this class can therefore determine the type of serialization a stream is bound to.

[3]Note that it is done also if an exception occurs while instantiating the user stream identifier class.

4.3 Serialization Awareness and Control

All the above relies on the idea that a metaobject is:

- aware of the fact that serialization is occuring;
- aware of the type of the stream used for serialization;
- able to control how its base object is serialized.

The first two points are referred to as *serialization awareness* and the last one as *serialization control*.

4.3.1 Serialization in Java

The Java programming language supports Object Serialization [58], that is the translation of any Java object to a sequence of bytes that can be reused afterwards to rebuilt the object completely. The sequence of bytes can be written to disk, to achieve persistence, or can be transmitted over the network (this is the marshalling/unmarshalling of arguments and return values in remote method invocation, *e.g.*).

By default, a Java object is serialized completely, meaning that all its fields are recursively serialized. In remote method invocation, the only modification to this principle is that if an object to serialize is a remote object, then a stub of this object is serialized instead of the object itself.

In any case, Java offers different means to specialize the serialization of an object. One mean is to make a class define two special methods, namely `writeObject` and `readObject`, in which one can respectively define the data that is effectively written to an output stream and how that data is read in order to build the object back. The limitation of this specialization mechanism is that a class can only control how members it *declares* are serialized, it has no control over inherited members.

Another way of specializing serialization is to use *object replacement* and *object resolving*. With object replacement, a class can specify an alternative object to be serialized instead of its instance. When deserialized, this alternative object must be resolved to an object that is type-compatible with the original object. This mechanism allows for much more flexibility and control over the serialization process.

As far as Reflex is concerned, the latter mechanism provides with means to achieve both serialization awareness and control, whereas the first one only allows for serialization awareness. It does not provide sufficient control over the serialization process for what we aim at.

4.3.2 Serialization Awareness

As mentioned above, serialization awareness is two-fold: being aware that serialization occurs on the base object, and being aware of the stream used

for that serialization. Making a metaobject aware that serialization is occuring is fairly simple: it just needs to be aware of the name of the public method invoked, what we refer to as the *serialization hook* method.

However, an implementation complication comes from how Java serialization works. Sequentially, when an object is to be serialized, the following happens:

1. the class is introspected to check whether it declares the object replacement method (namely `writeReplace()`). If it does, then the serialization process starts again but this time for the object returned by the replacement method.

2. the serialization of the object effectively starts.

3. the class is introspected to check whether it declares the `writeObject(ObjectOutputStream)` method, which allows for customized serialization of class-specific data. Note that at that point, when this is checked in the reflective class, all the data inherited from its superclasses has already been written to the stream.

A metaobject should be aware of the serialization stream used, and at the same time make use of object replacement (we will discuss this necessity in the following section dealing with serialization control). But, the `writeReplace()` method does not take as argument the stream used for serialization. Thus at that time, when this method is invoked, the metaobject does not know the stream used. Reciprocally, when the `writeObject` method is called, it is too late to use object replacement.

The solution to this issue is exposed thereafter, since it is strongly coupled with achieving serialization control.

4.3.3 Serialization Control

To be able to implement any semantics, a metaobject should be able to *control* how the serialization of its base object occurs. In fact, depending on the semantics being implemented, there are two cases:

The whole base object should be serialized. This case corresponds to backup serialization, and particular semantics of parameter passing or migration serialization. For instance in the default by-copy semantics, the whole base object has to be serialized.

The data of the base object should not be serialized. This case corresponds to particular semantics of parameter passing or migration serialization. For instance, in a by-reference semantic, the base object should be transfered empty. In those cases, only the metalevel of the base object should be serialized, so that it can control the semantics

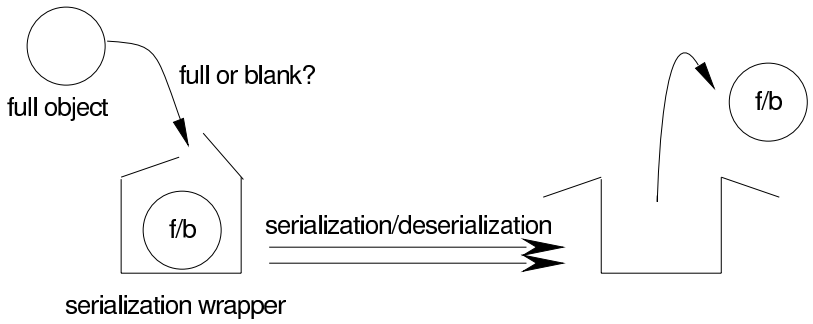

Figure 4.2: Principle of a serialization wrapper.

by forwarding method invocations to the appropriate object. All the other fields of the object should be empty (null), in order to make the object as lightweight as possible.

In the following, we refer to a normal object (the first case) as a *full object*, and to its lightweight equivalent (the second case) as a *blank object*.

The necessity of using object replacement is that it is the only way offered by the Java serialization API to control what is serialized, beyond the limit of the class-specific members. It is thus necessary to be able to transmit blank objects instead of full ones when needed.

4.3.4 Serialization Wrapper

Our solution to the two issues of full serialization awareness and serialization control relies on the introduction of a new kind of object, a *serialization wrapper*. The principle of such an object is illustrated in Figure 4.2.

When a full reflective object has to be serialized, a serialize wrapper is returned instead. This is implemented by the metaobject itself. The wrapper has a reference to the full reflective object it is replacing. Later, when the `writeObject` method is invoked on it, it queries the metaobject in order to know which object should be serialized. The metaobject looks at the stream being used, and returns a boolean indicating in which case we are: serializing the full object, or a blank object.

The serialization wrapper, possibly after creating the blank object, keeps only a reference to the object that should be serialized. The couple serialization wrapper–reflective object (blank or full) is then serialized, and when deserialized, the serialization wrapper is resolved to its embedded reflective object.

Briefly summarized, here is how serialization wrappers provide a solution to the issues of serialization awareness and control:

serialization is occuring The metaobject traps the invocation of the replacement method on the reflective object[4].

type of the stream used If the metaobject decided that it needs to be aware of this, it will have returned a serialized wrapper when the `writeReplace` method was invoked. In which case the serialize wrapper, when serialized, will invoke an extra MOP method to query the metaobject. This extra MOP method has been added transparently, using the metamessage feature of Reflex (see section 3.2.4). The metamessage sent takes as parameter the stream actually used for serialization.

control the object that is serialized By responding to the metamessage, the metaobject indicates to the serialize wrapper which object should be serialized, the full one or the blank one.

The concrete class of the serialization wrapper is chosen by the metaobject itself, since the latter has the control over which object is used for replacement. Therefore this framework for serialization awareness and control is freely adaptable.

As far as our implementation is concerned, we have developped a class of serialization wrapper, `dsm.SerializationWrapper`, that some of our metaobjects for data space management policies (part of the `dsm` package) make use of.

Creating and transmitting a blank object

As introduced earlier, a blank object is a reflective object that has the same metalevel part than its full equivalent, but whose fields are all set to null (object references and arrays). This way, serializing a blank object does not entail serializing all the data contained in its full equivalent. For instance, a vector keeps its data in a particular private field. In the blank object equivalent of that vector, this field will be set to null, entailing that all the objects contained in the vector will not be serialized.

The main idea of a blank object is to externally be a normal object, but as lightweight as possible, with a metalevel that takes care of implementing the correct behavior. Such an object is needed to simulate a by-reference semantics, for instance.

A blank object is created by instantiating the reflective class, properly initializing the metalevel fields (such as the reference to the metaobject), and, using reflection, setting all the other fields (inherited) to null. Fields which are of a primitive type are unchanged.

Figure 4.3 illustrates the exact process that occurs when creating and transmitting a blank object. The original situation is a full reflective object

[4]More details are exposed in section 4.3.5.

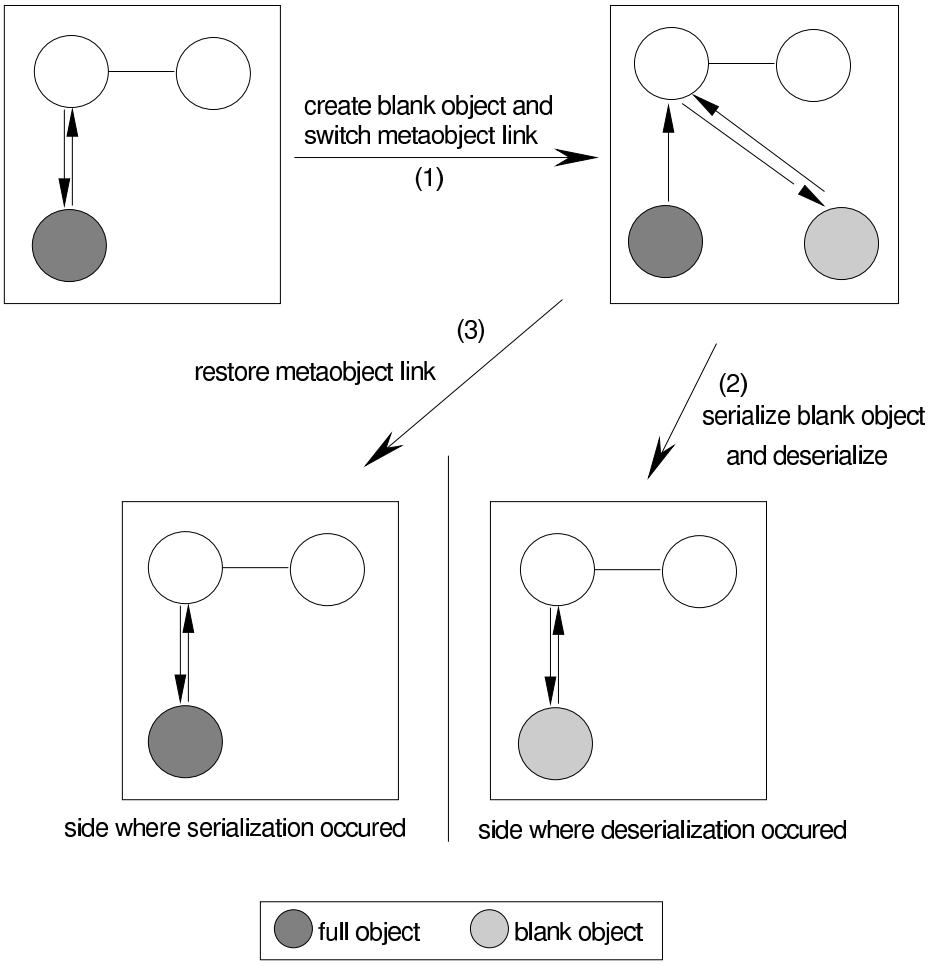

Figure 4.3: Creating a transmitting a blank object—the necessity of switching the metaobject link.

with some attached metaobject. Then, the blank object is created, pointing to the same metaobject as the full object. However, for the blank object to be in a consistent state when deserialized, the link from the metaobject (the root metaobject) to its base object should be updated so that it points to the blank object (step 1). Then the blank object is serialized, and deserialized in a consistent state (step 2). As soon as serialization finished, the metaobject link is re-established as it was before, in order to restore the initial state (step 3).

This way, we have on the side where serialization occurs the same situation as before, and on the side where deserialization occurs, we have exactly the same configuration, but with a blank object instead of a full one.

4.3.5 Extension of the Reflex Framework

The Reflex framework has been extended to include an appropriate class builder able to build reflective classes ready for serialization awareness and control, as well as a new abstract metaobject class that combines the stream identification facilities with serialization awareness and control.

The SerializeReflexObject interface

In the wrapping mechanism previously exposed, there is an obvious risk of infinite recursion: a reflective object gives as replacement for itself an object that points to it. Thus the serialization process will enter in an infinite loop.

Therefore, a reflective object ready for serialization should have an extra state information that indicates whether it is wrapped or not. If it is wrapped, the replacement method returns itself directly. If it is not wrapped, the replacement method calls a public serialization hook that the metaobject will trap (to return, possibly, the serialization wrapper). Similarly, a blank object should never be wrapped. Therefore another state information is added to reflective objects, indicating whether it is a blank object or not.

To this end, we have introduced a new interface for reflective object that are ready for serialization, the `reflex.SerializeReflexObject`, a subinterface of the `ReflexObject` interface. It adds accessors to state indicators (see table 4.2).

The SerializeClassBuilder class

In order to build reflective classes with serialization awareness and control, we have defined the `reflex.builder.SerializeClassBuilder` class, a new class of class builders, subclass of the `BasicClassBuilder` class.

A `SerializeClassBuilder` object first lets its superclass build the reflective class, and then does the following:

- makes the class implement the `java.io.Serializable` interface;

method name	description
`Boolean _isWrapped()`	returns true if the object is already wrapped.
`void _setWrapped (Boolean b)`	sets the wrapped flag to the given boolean.
`Boolean _isBlank()`	returns true if the object is a blank object.
`void _setBlank (Boolean b)`	sets the blank flag to the given boolean.

Table 4.2: The SerializableReflexObject interface.

- makes the class implement the `reflex.SerializableReflexObject` interface;
- adds two boolean instance variables, `blank` and `wrapped`;
- adds the four methods of the `SerializableReflexObject` interface. The implementation of those methods in the method factory uses the previously added boolean instance variables;
- adds the serialization methods:
  - `writeReplace()`—object replacement method from the Java Object Serialization framework. The implementation of this method first checks if the object is blank or wrapped, in which case it simply returns `this`. Else it calls the public serialization hook below;
  - `doWriteReplace()`—public serialization hook that informs a metaobject that serialization is occuring, and lets it return a possible alternative object for serialization. The implementation of that method simply returns `this`, in order to remain compatible with metaobjects that do not listen to serialization.

When this is done, the class is returned and the building process goes on, making the class reflective and persistent on disk, before instantiation.

There is actually a major issue with this process concerning classes that cannot easily be made serializable. It is exposed in details in section 4.3.6.

The MigrationMetaobject class

The `reflex.metaobj.MigrationMetaobject` class is an abstract class that extends the `StreamMetaobject` class. It thus has the stream identification facilities provided by that class.

In addition to that, this class specifies that the class builder it requires is a `SerializeClassBuilder`, since its instances will make use of serialization awareness and serialization control.

Therefore, instances of concrete classes of this class will be ready to operate in a distributed environment with migration: they have the abilities to determine whether migration is occuring or not and control the transmission of their base object. Indeed, this class served as the base class for the different classes of metaobjects we have implemented for resource management policies, presented in the next chapter.

4.3.6 Advanced Serialization Issue

There is an issue that we have overshadowed until now: the problem of making non-serializable classes serializable. Recall that in section 4.3.5 we have exposed that a `SerializeClassBuilder` makes a reflective class *serializable—i.e.* it makes it implement the `java.io.Serializable` interface.

The `Serializable` interface is just a marker interface for which the serialization process checks for in order to go on. If it encounters an object to serialize whose class does not implement this interface[5], it will stop and throw an exception.

The issue

For some classes, simply adding the fact that they are serializable will not entail any problem, in particular if they only declare instance variables which are of a serializable type. However, if a class declares instance variables which are not of a serializable type, then an exception will be thrown when they will be serialized. This occurs only if the object is serialized completely (full object), of course—since if it is a blank object, the fields of the object are all null. In the case where the associated metaobject orders a serialization of the meta-level only, there will not be any problem... except if the class of the instance to serialize does not respect the following law:

> "a `Serializable` class must be able to access the no-arg constructor of its closest non-Serializable superclass." [57]

This is actually where the big issue lies. Because the problem of not being able to take along a non-serializable object completely is obvious. But, there are cases where the whole object is not serialized, just its meta-level part, and still there can be a problem: *the serializable class needs a non-serializable superclass that declares an accessible no-arg constructor.*

An example

Let us consider a by-reference alike semantics. A mobile agent has a reference to a `PrintStream` and moves to another site. The `PrintStream` is in fact a reflective object to which is attached a by-reference metaobject.

[5] or any of its subinterfaces, like the Externalizable interface.

Thus, when migrating, the metaobject specifies that the content of the object should not be serialized, since once in the new site, it is the metaobject itself that will forward requests to the `PrintStream` that stays in the first site. What is expected from this case is that when the agent prints on the remote site to the `PrintStream`, it will in fact print on the local one.

However, this will not work since the `java.io.PrintStream` class does not have an accessible no-arg constructor. This problem is the same with most of the output and input stream classes of Java. Though it is semantically possible, it does not work because of this limitation.

A solution

Apart from rewriting the serialization process entirely, there exists an alternative to solve this issue. As a matter of fact, in semantics such as the one exposed in the example above, the actual object that is deserialized does not play an important role: only do we need it for its public interface, since then it is the metaobject that handles the rest. Therefore if we can find a way to build a dumb object, yet valid, it will solve the problem.

The idea is to provide a no-arg constructor for such a class, that would call a *true* constructor with the required parameters. For instance, in the case of the output streams of Java, they can all be created by specifying an output stream on top of which they operate. You could then add a no-arg constructor that will call the normal constructor with as argument a minimal output stream, such as a `ByteArrayOutputStream` of size zero. The same "trick" would work similarly for all the input stream classes of Java, making them read from an empty `ByteArrayInputStream`.

In our example above, the `PrintStream` on the remote host would in fact be a PrintStream on an empty `ByteArrayOutputStream`. As it is a valid object, the mobile agent can still print to it. Since all calls are trapped by the metaobject and forwarded to the local `PrintStream`, the dumb object is in fact never used: nothing will be printed to the empty `ByteArrayOutputStream`.

Implementing the solution – serialization adapters

The actual solutions to this issue are very case-specific. It is impossible to specify in a generic way how a no-arg constructor should be created. However, it is yet possible to make this process flexible, and reusable by optimizing the possibilities of "making the trick for several classes in one shot".

When the `SerializeClassBuilder` is asked to build a reflective class, before doing anything, it checks whether the class is problematic, *i.e.* if (1) it is not serializable, and (2) it does not have an accessible no-arg constructor. If it encounters such a problematic class, it does the following:

- creates an intermediate non-serializable subclass of that class, adding to it a no-arg constructor;
- creates a serializable reflective subclass of that generated class (this class will be the one returned to Reflex for instantiation).

Creating a class and adding a constructor to it is straightforward with Javassist. Then making a reflective subclass of it is what we have already explained. Thus the only issue is from where the no-arg constructor should be copied (recall that Javassist only allows for bytecode-copying of methods and constructors).

In fact, the no-arg constructor is copied from what we call a *serialization adapter* class. An adapter class has the following characteristics:

1. it has the same name than the class it is adapting, within the package `reflex.adapters`;
2. it is a class (possibly abstract) that extends the adapted class;
3. it has a public no-arg constructor that calls `this(<args>)` (it should not call directly `super(<args>)`)[6];
4. it has another constructor, the one called from the public no-arg constructor, which makes the super call[7].

When the `SerializeClassBuilder` is building the intermediate subclass with a public no-arg constructor, it looks for an adapter to the class it is working on, and if it cannot find it, it attempts to get an adapter for its superclass, etc. until it gets to the `Object` class (in which case it throws an exception because it cannot go on). The main point of this mechanism is to allow a user to specify an adapter for a complete subtree of the class inheritence tree, while still allowing for specific specializations where needed.

In our example, we could define an adapter for the `PrintStream` class, by defining the `reflex.adapters.java.io.PrintStream` class. However, since `PrintStream` is a subclass of `FilterOutputStream`, we can directly create an adapter for that class—we know that all instances of this class and its subclasses can be created by giving an empty `ByteArrayOutputStream` as argument to the standard constructor. This way, if ever we need to use an `ObjectOutputStream` for instance, the adapter will automatically be found and used by the class builder.

The code below defines a serialization adapter class for the output stream abstract class `java.io.FilterOutputStream`—and all its subclasses.

[6]This comes from the fact that the constructor is bytecode-copied and then inserted into the new class: super references do not behave correctly in that case. The class would be rejected by the bytecode verifier of the JVM.

[7]This constructor is never copied and inserted into the new class, it just serves to produce correct bytecode for the no-arg constructor.

```
package reflex.adapters.java.io;
import java.io.*;

public abstract class FilterOutputStream
    extends java.io.FilterOutputStream {

    //the public no-arg constructor that will be copied
    public FilterOutputStream(){
        this(new ByteArrayOutputStream(0));
    }

    //the needed dumb constructor
    FilterOutputStream(java.io.OutputStream o){
        super(o);
    }
}
```

With such a small class defined, it is now possible to get reflective stream objects and use them in several semantics of resource management, like in the example above, or as illustrated in the next chapter. Similarly, on a case-by-case basis, but still with some genericity, it is possible to overcome the limitation exposed in this section.

4.4 Summary

In this chapter we presented the extension of Reflex to make it fully operational in a distributed environment, especially within a mobile object system.

We first discussed the broad idea behind this adaptation, the one that made us think about a system like Reflex as a solution to achieve flexibility in resource management.

We talked about the minimalistic interface needed between Reflex and any Java mobile object system, and detailed the Reflex extension that simplifies integration.

Finally we introduced the concepts of serialization awareness and control, the necessary cornerstones of Reflex operatibility in distributed systems, and explained how they were added to Reflex.

Now Reflex is up-and-ready for our target application domain. We can start using it to implement different semantics of resource management.

Chapter 5

Metaobjects for Resource Management

The original aim of this thesis was to provide flexibility in resource management, in the particular area of mobile object systems. Until now, we have presented the Reflex system as well as its extension to the world of distributed computing.

In this chapter, the practical use of Reflex for our target domain is illustrated through concrete examples. The development of two classes of metaobjects implementing two different resource management policies is exposed, one for the *network reference* policy and one for the *rebinding* policy (see section 2.1.1). Sample test programs exercising these metaobjects are also included.

5.1 Introduction

To apply and thus validate our ideas, we developed metaobjects for Reflex that implement different resource management semantics. We chose to experiment with the network reference and rebinding policies, since we see them as the most interesting in practice—the by-copy semantics being already available.

Numerous examples of applications of these semantics can be found, especially in the area of code mobility, or more precisely in mobile agents. For instance, the network reference policy is useful in cases where a unique object has to be accessed by several mobile entities, keeping consistency at any moment, or in cases where the tradeoff network reliability/bandwidth indicates that it is better for a somewhat huge object not to be transmitted, but instead to be accessed remotely. The rebinding policy as well is useful in all cases where a roaming entity repetitively needs to access a resource that is local to the host, such as I/O resources for instance.

These two policies have therefore been implemented, and tested with

different types of resources. To perform experiments, we wanted a mobile agent platform. The problem was that it needed to be a platform compatible with the JDK1.2, since Javassist relies upon it. Unfortunately, the Aglet platform has not been updated within the last year, and is thus inadequate for the JDK1.2. Finally, after looking in vain for a compatible and working Java mobile agent platform, we undertook the development of our own, the EzAgent platform[1]. This platform is an experimental platform relying upon RMI[2]. Only agent migration is implemented, since it was the only required feature for our tests. This means that communication inter-agents has not been implemented. This platform is presented more precisely in Appendix B.

The metaobject classes we have developed are part of the package `dsm`, which stands for data space management. The package `dsm.rebind` contains the classes for implementing the rebinding policy, and the ones for the network reference policy are in the package `dsm.remoteref`. The metaobject classes, `RebindMetaobj` and `RemoteRefMetaobj` are subclasses of the Reflex metaobject class `MigrationMetaobject` (see section 4.3.5).

5.2 The Rebinding Policy

To achieve the rebinding policy, we have to assume that hosts on the network provide a way of getting a reference to a local resource based on some identifier. This way an agent can rebind this resource each time it arrives to a host, and unbind it each time it leaves.

There are several ways such a system for local resources management can be implemented. As for the tests, we have developed a very simple resource management system, that allows any program to bind a resource to a string identifier. Any incoming entity can query the resource manager in order to get a reference to this previously bound resource. In order to get the resource manager, we have extended the `java.lang.System` class with an extra method, `getResourceManager()`, that returns a reference to the current manager. There is one resource manager per JVM. More details on the resource manager can be found in Appendix C.

The main point of using Reflex for this purpose is that the actions of rebinding and unbinding are handled automatically by a metaobject. The object that holds a reference to such a resource just needs to instantiate the resource using Reflex, specifying an instance of our class of metaobjects for rebinding as parameter. Then it simply accesses the resource as it would do normally, without having to implement the rebinding and unbinding operations.

[1]The *Ez* prefix (easy) was inspired by the basic and limited nature of the platform.
[2]Therefore the work to integrate Reflex into EzAgent was reduced to nothing.

5.2.1 Design and Implementation

The design principle is illustrated in Figure 5.1. The agent holds a reference to a dumb reflective object of the appropriate type. This dumb reflective object has an attached metaobject that maintains the binding to the concrete local resource (obtained from the local resource manager). Each invocation on the dumb base object is trapped by the metaobject that forwards it to the bound resource. Thus the dumb object is concretely never used, it is the resource bound to the metaobject that is used instead. This way, the agent always invokes methods on the same object, and it is the metaobject that takes care of forwarding those calls to a local resource (Figure 5.1(2)).

The metaobject is responsible for making the binding to the resource. Obviously, it needs to know the identifier to be used for the resource manager. In our implementation of the rebinding metaobject, `RebindMetaobj`, this can be specified when the metaobject is first created, or later, using a special metamessage. We have implemented the initialization of the resource in a lazy-manner. When the metaobject traps an invocation on the base object, if the resource has not been bound already, then it does it by requesting the local resource manager (Figure 5.1(1)).

Unbinding the resource is done each time the base object is serialized. This is systematic, since generally a local resource is not something one wishes to serialize, even for a backup. It is more a local service accessible to any object that requires it. Simply, each time the metaobject itself is serialized, it unbinds the resource. This is implemented by declaring the resource object as a *transient* field of the metaobject.

For performance optimization, the metaobject keeps a cache of the methods of the resource in an array (this array is also a transient field). Method invocation is then performed using the reflection API of Java. Note also that all methods invoked on the base object are forwarded to the resource, *except* the serialization request.

5.2.2 Examples

Let us consider a simple test case: a roaming mobile agent has to perform some activity in several nodes on the network, following a given itinerary. For tracking purposes, we want this agent to log its actions into a file. Each site on the network has a local resource named `logfile` in which such roaming agents can write.

To write to a file, the agent has an instance variable of type `PrintStream`, a convenient class of the Java I/O library for writing text, and a method that logs a given operation in the log file:

```
// declaration of the instance variable.
PrintStream logfile;
```

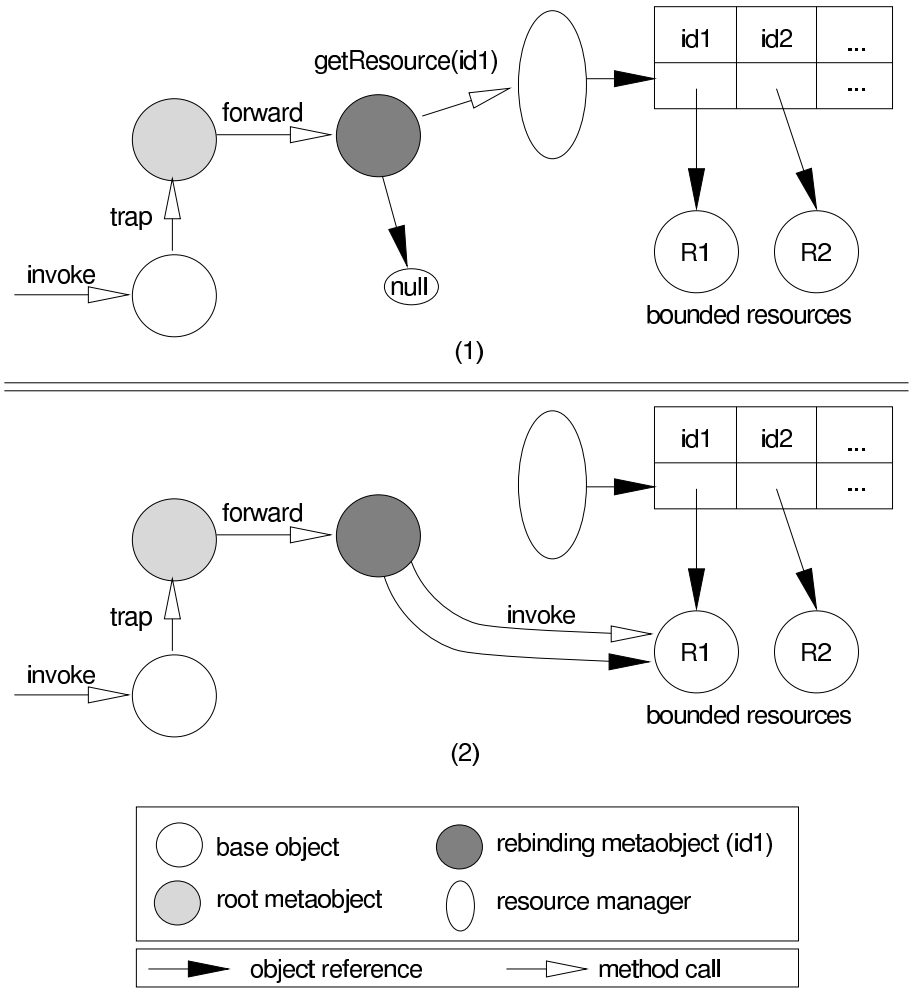

Figure 5.1: Design of the rebinding policy—(1) Lazy-initialization of the resource. (2) The resource is initialized, calls are forwarded to it.

```
//log method that uses the instance variable.
private void logWork(Operation op){
    logfile.print(this + `` has performed: '');
    logfile.print(op.getDescription());
    logfile.println(``on date: '' + new Date());
}
```

There are different alternatives for initializing the `PrintStream` resource, either in the code of the agent itself, or it can be passed as a parameter to some method, the constructor for instance. In any case, the following code creates a dumb `PrintStream` with the correct rebinding metaobject attached:

```
// create the metaobject, specifying the resource identifier.
ReflexMetaobject mo =
    new dsm.rebind.RebindMetaobj(``logfile'');

// instantiate the PrintStream object (dumb).
logfile =
    (PrintStream) Reflex.createObject(``java.io.PrintStream'',
                                      mo);
```

Note that the created object is a dumb object, instance of the class created by Reflex using the appropriate serialization adapter (see section 4.3.6), since `PrintStream` is normally a class that cannot be serialized.

We have successfully implemented this example using the EzAgent platform, with two different kinds of resource: a simple Java vector and a `PrintStream` to a file, exactly like in the example above. More precisely, our agent is created remotely in a given site, the resource being passed as a parameter, writes something, moves to another site, and writes the same thing again. Each agent place (EzPlace) has been extended to automatically register the local resource with the resource manager when it is started. Also, in order to validate the serialization identification mechanism, each agent went for deactivation (serialization to disk - timer - deserialization) in each site.

This example illustrates how Reflex largely simplifies the implementation of a rebinding policy. Of course, this policy can be implemented without Reflex, putting all the code specific to binding/unbinding of the resource in the agent itself [3]. Anyhow, it is a fact that this code *is not specific* to the agent, and will have to be copied to any other object that requires the same semantics, solution which is highly inflexible and unmaintainable. Here we achieve a convenient separation of concerns, avoiding to mix application code with resource management code, as well as enhanced maintainability and reuse.

[3] Though the issue of the non-serializable classes remains.

5.3 The Network Reference Policy

The Java RMI mechanism already provides a mean to achieve a by-reference policy. An object that has to be passed by reference has to be of a remote type, and then, when serialized, a stub to the object is passed instead. Then, invocations on the stub are forwarded to the remote object.

The problem with this mechanism is that it is far from being flexible. The class of remote object has to implement a sub interface of the `RemoteObject` interface, and must extend another class of the RMI framework. In addition to this, stubs and skeletons have to be generated manually. All this is static, it has to be done at implementation time. There are no means to make a non-remote object remote dynamically.

However, the basics of the by-reference semantics are implemented, and can be used to implement a more flexible mechanism. Reflex can be adequately used for that purpose.

5.3.1 Design and Implementation

As mentioned before, the main points of a by-reference semantics are first to avoid transmitting a huge object over the network, and second, to maintain consistency between different accessors to that object.

To avoid transmitting a huge object is quite straightforward: the metaobject should specify that a blank object should be transmitted instead of the whole object (see section 4.3.4).

Then, to forward the method invocations to the local object, we already saw in the previous section presenting the rebinding policy that a metaobject can fairly easily forward calls to any other object instead of its base object. To be able to forward calls to the local object would mean that the local object should be a remote object. However with RMI this is not flexible at all. The idea is to introduce a special kind of object, a *remote proxy* object. Such an object is an RMI remote object, but a generic one: it is a method invoker, able to invoke any method on any type of reflective object. Therefore, the metaobject of the blank object will forward all method invocation to this remote proxy (that resides on the same site than the local object), and this remote proxy will in turn invoke the method on the base object.

Architecture

The architecture is illustrated in Figure 5.2. On site A, a reflective resource has an attached metaobject of class `RemoteRefMetaobj`. When first serialized for migration, the metaobject sets up the remote proxy, and serializes itself with a reference to that proxy. Since it is a remote object, when the metaobject will be deserialized, it will have a reference to a stub of the remote proxy, and will thus be able to perform remote method invocation on

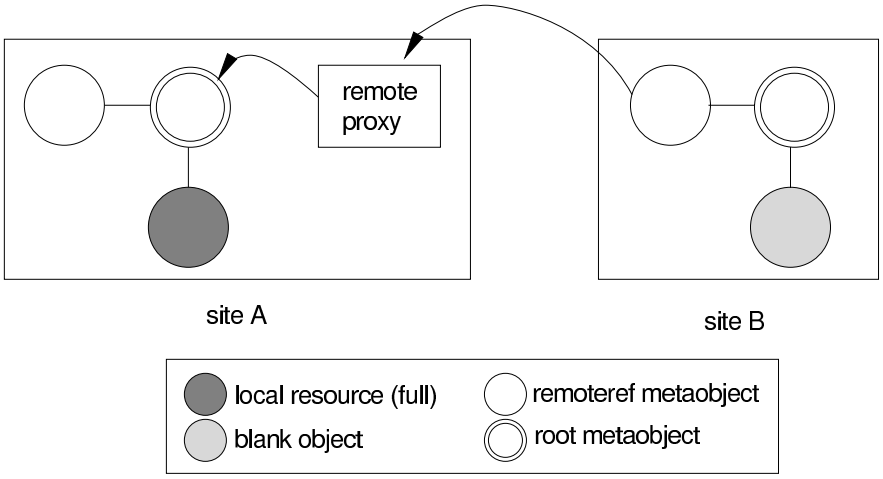

Figure 5.2: Design of the network reference policy.

it. Apart from setting up the remote proxy, the metaobject also specifies that the base object should not be transmitted fully, a blank object should be created instead.

Therefore on site B it is a blank object of the same type that has a metaobject that will forward any invocation to the remote proxy. In order to invoke methods on the resource, the remote proxy is initialized with a reference to the root metaobject of the resource. Recall that the root metaobject offers the service for invoking a method on the base object.

Using this metaobject, it is possible to make any type of resource accessible by the network reference policy. Nothing needs to be specified statically in the class of the resource, only does it need to be instantiated or converted using Reflex. Since the remote proxy is a generic one, stubs and skeletons have been generated once for all, and thus there is no need to perform any compilation manually.

Implementing the metaobject

Implementing the metaobject for network reference is slightly tougher than implementing one for the rebinding policy since the metaobject does not always behave similarly. In the rebinding policy, every metaobject has the same behavior, may it be the first metaobject, or the one recreated after transmission. In the network reference policy, however, there are two logical states for the metaobject:

local In this state, the metaobject should set up the remote proxy when serialized for migration, if it does not already exist. In addition to this, it should serialize a version of itself which is in the remote state.

All method invocations are simply invoked on the base object.

remote In that state, the metaobject should serialize itself as it is, with no modification to its state. All method invocations, except serialization, should be invoked through the proxy and not on the base object.

To implement these states, we have applied the *State* design pattern [34]. A `RemoteRefMetaobj` object aggregates a behavior object, instance of the abstract class `RemoteRefBehavior`. There are two concrete subclasses of this abstract class, `BehaviorRemote` and `BehaviorLocal` that implements the actual behavior to be done. A `RemoteRefMetaobj` delegates requests to its aggregated behavior object.

As pointed out in [11, 34], the *State* design pattern lets open the issue of where the state switching occurs. We have implemented it at the level of `RemoteRefMetaobj`, in the customized serialization method `writeObject`. If the metaobject is in the local state and goes for migration, then the behavior object that is written to the stream is an instance of `BehaviorRemote`.

Remote proxy and concurrency

Since one of the advantage of the network reference policy is when several entities should access the same object, it is necessary to take into account the issue of concurrency at the level of the remote proxy.

For that purpose we simply implemented a second class of remote proxy, whose public method is synchronized. However it is not always needed to have a synchronized remote proxy, either because the base object itself is synchronized already, or because only one object will access the resource.

To let the programmer able to specify what he wants, our implementation of the network reference metaobject understands a specific meta-message, `setSynchronizedProxy`, that can be used to specify which proxy class should be instantiated. Note that if the remote proxy has already been created, it is not possible anymore to change its type. This can also be specified as a boolean argument given as a parameter when instantiating the metaobject.

5.3.2 Examples

Let us consider the following test case: in the field of information retrieval, several agents are launched on the network with the task of getting some information. Each time an agent finds a relevant piece of information, it should store it in some vector. In order to take advantage of the network reference policy, let us say that a unique vector is created and used by all agents to store data. This way, when the user wants to check what information is currently available, he just has to look at the elements of the vector, he does not need to contact each and every agent.

The central vector will be created either by a client application, or by a coordinating agent that in turn creates several slave agents to look for information. In any case, the following code will instantiate the result vector:

```
// create the vector with a metaobject for network reference.
// the true argument passed when instantiating the metaobject
// informs that the remote proxy should be a synchronized one.
Vector results =
    (Vector) Reflex.createObject(``java.util.Vector'',
                                 new RemoteRefMetaobj(true));
```

Then, the program that looks up the state of the vector just accesses it as usual:

```
public void displayResults(){
    System.out.println(results.toString());
}
```

And the roaming agents themselves add elements to the vector in a fully-transparent way. They do not even need to know that the vector they are adding elements to is a remote one.

```
//instance variable declaration
Vector myResults;

//method that adds to the result vector
protected void addInfo(Information info){
    myResults.add(info);
}
```

Like for the rebinding policy, we have successfully implemented this example using the EzAgent platform, with a Java vector like above, and with a `PrintStream` to the JVM screen. The simulation was with three roaming agents, accessing the same "by-reference" resource, while moving from one host to another. As in the previous test, each agent went down for deactivation on each site, thus testing the serialization identification abilities of Reflex.

We also carried out another simulation, this time combining the two policies exposed: each agent first writes something to a resource by reference and then to a similar resource managed with the rebinding policy. This example corresponds to cases where roaming agents log information to each node as well as to their starting node (*e.g.* remote administration).

Here again we have illustrated how Reflex simplifies the implementation of a resource management policy. Using the `dsm.remoteref` package, a programmer can benefit from a network reference policy without having to generate skeletons, adapt all the class hierarchy, and so on. The example makes full use of the transparent type-compatibility of reflective objects to achieve a clean separation of concerns.

5.4 Summary

In this chapter, we have validated the concrete applicability of Reflex to the area of resource management in mobile object systems.

We have exposed the design, implementation and testing of two kinds of metaobjects: one for the *rebinding* policy, and one for the *network reference* policy.

In each case, a short test case has been presented as well as results of the simulations carried out during the thesis. These tests have shown the correctness and feasibility of the semantics simulated.

The tests also demonstrated the functioning of the interface between Reflex and mobile object systems, since different types of serialization were used and properly exploited.

We therefore validated our starting assumption according to which the different semantics of resource management could be reasonably easily implemented as metaobjects attached to the resources.

Here ends the presentation of the concrete work done during this thesis. Now begins our own evaluation of the work, perspectives for further research, and conclusions.

Chapter 6

Evaluation

In this chapter we perform an evaluation of the work carried out during this thesis. In particular, we examine the performance issues related to Reflex and the current limitations of Reflex.

6.1 Achievements

During this thesis we have developed the reflective system Reflex. This system meets our four primary objectives:

- it provides *transparent type-compatibility* of a reflective object with its non-reflective equivalent. This was achieved by making a reflective object instance of a reflective subclass of the class we want some reflective objects of. Type-compatibility is therefore achieved by subtyping (see section 3.2.1).
- it is able to *create* reflective objects and *convert* existing non-reflective objects to reflective ones. Since conversion is achieved by creating a whole new object, there is the limitation that such a conversion should be done before any references to the object are passed to the "outside world" (see section 3.2.2).
- it is possible to *dynamically control* the layer of meta-behaviors associated to a reflective object, thanks to the metaobject composition framework we have set up (see section 3.2.3).
- it is sufficiently *open* to be adaptable to unforeseen uses, since we have let an open door for customization of the MOP with the concept of metamessages (see section 3.2.4).

Moreover, we have successfully extended Reflex, which is a priori application domain-independant, so that it is operational in a distributed environment, making its interface with mobile object systems minimal (see

chapter 4). Finally we have concretely applied the extended Reflex to implement different resource management policies and tested the same with mobile agents (see chapter 5).

The Reflex system as well as the different metaobjects developed, including sample test programs, will soon be available in the Internet for download. Until then, they can be requested to the author by e-mail at `etanter@vub.ac.be`.

6.2 Performance

Using Reflex is not free in terms of performance. There are several points where extra processing is introduced:

Cost of meta-level processing. Having an interpretation layer at the meta-level is obviously a loss of performance. However it is important to see that this loss is reduced to some extra method calls. This cost is relatively negligible when used in a distributed environment, where real performance problems come from the network latency time [22].

Cost of reflection. In Reflex many basic operations rely upon the reflection API of Java. For instance, method invocation on the base object is performed using reflection, and the class building process uses some reflection as well. There is a cost for using reflection, that cannot be avoided. However, some research is actually being done to optimize the use of reflection in Java, thanks to partial evaluation techniques [12]. A system like Reflex can hope to benefit from results of this research area.

Making reflective classes. The whole process of making a new reflective class is also time- and resource-consuming. Moreover, this can happen at run-time since Reflex classes are created in a lazy manner. However, once generated, a reflective class is written to disk. Therefore the generation cost is restrained to its minimum. Once on disk, a reflective class is yet another Java class.

Cost of the meta-level architecture. The meta-level associated to a reflective object is at least of one metaobject, the root metaobject, plus all the effective metaobjects. This entails some cost since metaobjects collaborate together. It is a price to pay to have a composition framework and to allow dynamic composition/decomposition of metaobjects. We believe it is worthwhile, since the flexibility offered can adequately be exploited and since Reflex was targeted at distributed environments.

Having reflective objects. With Reflex, only instances which are *intended to be used as reflective* are reflective, since the base class is not modified. Compared to a classical approach, having one reflective object in a system does not entail that all other objects of the same type are reflective too. Therefore the cost of having reflective objects (meta-level processing) is reduced to only those objects that really need it.

Of course, using Reflex will never make a system faster. There is an inevitable tradeoff between flexibility and performance that has to be evaluated on a case-by-case basis. We have made our best to limit the over-cost of Reflex, as explained above, but it will ever remain.

6.3 Limitations

6.3.1 Final classes

The base principle of Reflex is to create a reflective subclass of the class we want reflective instances of. This principle encounters a limitation, however. In the Java programming language there is a modifier, named `final`, which aim is, when applied to a class declaration, to disable the possibility of extending the class. Such a class is referred to as a *final class*. Though it is still possible to *create* a reflective subclass of a final class, it will be rejected when *loaded* by the bytecode verifier of the Java virtual machine.

Therefore, it is currently impossible to get a reflective instance of a final class, though semantically it should be possible. Indeed, what Reflex does is not modifying the semantics of the class itself, but instead adding the possibility of customizing it. However, for a first approach, we have found this limitation not too dramatic, since very few classes are declared as final classes in the Java library. Moreover, most of the classes that are good candidates for our target application domain, in particular for resource management, are not final classes (*e.g.* all classes of the `java.util` package, all classes of streams, sockets, etc.).

However, the limitation remains. There is actually one solution that could be implemented at load-time: using Javassist, it is possible when loading a class into the JVM to remove its final flag. In this way, if later on a reflective subclass is loaded, it will not be rejected by the bytecode verifier. However this implies using a customized loader for loading all classes included the system classes, and this is a very sensitive manipulation. Also, in a distributed environment, it implies being sure that all hosts have been started using the customized class loader. Finally, removing the final flag of classes introduces a priori a security hole in the system, since it would allow to load other, possibly non-trusted, subclasses of normally final classes.

6.3.2 Final methods

Similarly to the previously exposed limitation, there is an issue with public methods of a class that are declared `final`. Applied to a method declaration, this keyword indicates that the method cannot be overridden. Or, in order to be able to trap all public method calls, a reflective class overrides each and every public method declared in any of its superclasses.

Currently, if a final method is encountered, it is skipped, since it cannot be overridden. Therefore, invocations of this method, though public, will not be trapped by the metaobject. Again, in practice, we did not encounter any issue with this. But our level of experimentation remains very basic and it is possible that this limitation could be problematic.

Again a possible solution to this issue is using a customized class loader and Javassist. The idea would be to modify any class that has at least one final public method. For each final public method, we should rename that final method, following some internal naming convention, and then add a new non-final public method with the original name. This new method will simply delegate behavior to the original final method. Doing such a transformation is easy with Javassist, but like the solution to the previous issue, it requires acting on each and every class loaded into each JVM of the (possibly distributed) system.

6.3.3 Composition of metaobjects

Composition of concerns in general is a very tough topic. Different approaches have been explored by the research community ([10, 9]), but the issue is still open. In particular, correctly managing the possible cross-cutting and collaboration of concerns, may them be represented by metaobjects, or composition filters [4], or separately specified aspects [37], is still an active research topic today[1].

In Reflex, we have implemented a framework for metaobjects composition based on work on explicit composition of metaobjects [50]. In section 3.2.3 we have exposed it, as well as an indicative terminology for metaobjects, introducing the idea of a *semantically strong* metaobject, an *auxiliary* metaobject, and an *adaptor* to a semantically strong metaobject.

However, we feel that work remains to be done here to explore more deeply the composition model and its pitfalls. Surely this was not the aim of this dissertation, and it would require a considerable investment to explore this issue in details. Though the model was sufficient for our experiments, we are convinced it would need some rework, taking more time to study in depth the different works done in the field. In addition to that, it will be necessary to consider the issue of dynamic metaobjects composition with

[1] A sign of this fact is the presence of a workshop dedicated to advanced composition of concerns at the upcoming OOPSLA'2000 conference.

class builders, in particular the possibility of the case where a metaobject is added to an object that has been created using a class builder that is not specialized enough for that metaobject. Therefore we foresee that this part of Reflex is likely to be the most subject to changes in the future.

Chapter 7

Conclusions and Perspectives

7.1 Future Work

Concerning the Reflex system, we foresee three directions for future work. First of all, in order to solidify the Reflex architecture, it is necessary to develop more kinds of metaobjects for different semantics. For instance, it should be possible to extend our network reference metaobject to implement a transparent replication mechanism. Developing other metaobjects will further test-proof the architecture of Reflex, possibly entailing slight modifications.

A second point to focus on is Reflex integration to existing mobile object systems. During this thesis, we were limited in the sense that most mature mobile agent platforms for Java are only compatible with the JDK 1.1 whereas Reflex requires JDK 1.2. Most of the implementors of mobile agent systems plan to make a version compatible with JDK 1.2 in the near future, but as of now we have not encountered them. As soon as several platforms are ported to JDK 1.2, the Reflex integration mechanism should be tested, though we do not foresee real issues in that regard. Indeed, we believe that our interface is adequate to avoid the integration problems.

Finally, as mentioned in section 6.3.3, the framework for metaobjects composition requires a deeper analysis. More work is needed on composition semantics and how to guarantee a certain level of consistency when composing meta-level entities, addressing possibly cross-cutting concerns.

7.2 Perspectives

We believe that Reflex is particularly well adapted to achieve flexibility in mobile object systems. Indeed, its applicability can be extended: until now, we have only used Reflex to make passive objects reflective, and used that ability to implement different resource management policies. But Reflex in itself does not restrain its applicability to passive objects. Active objects,

agents, can also be made reflective. This possibility opens many interesting perspectives:

Alternative method implementations. Using an extended class builder, it is conceivable to allow the specification of different implementations for a particular method, and then make a metaobject invoking any of the available implementations given a selection criteria. This functionality can be implemented in a very flexible way using Reflex, since Javassist offers all the convenient methods to manipulate bytecode representation of methods. Such a functionality is useful in order to achieve dynamic behavioral adaptation of mobile entities: for instance, a mobile agent could use different compression algorithms depending on some external criteria, *e.g.* network congestion.

External resource awareness. Following the idea of Sumatra [1], it is important to be able to make a mobile agent aware of the state of the resources available, like the quality of the network at a given moment, or the state of a persistence device. In the rebinding policy, we used Reflex to transparently connect resources to a local resource manager (section 5.2). This resource manager was seen as a repository of references to local resources. Similarly, such a resource manager could be extended to provide notifications about the state of external resources. A metaobject could be implemented to make mobile agents themselves register with such a manager, and adapt their behavior accordingly.

Combination of both. In fact we foresee a combination of the two previously exposed functionalities, along with the resource management policies implemented during this thesis. A metaobject (or set of collaborative metaobjects) could use information from a resource monitor to determine which implementation to use for a given method. Using the resource monitor, the metaobject could also change the management semantics of some resources (passive objects) attached to the metaobject. This would require an extension of our metaobjects, in particular that of network reference, in order to make it possible to consistently alternate between a network reference and a by-copy semantics.

As of now we have quite a clear idea of how to implement these different features properly with Reflex. We did not have time to give it a deeper thought during the time of this thesis, but we plan to effectively do so in the near future. Indeed we are convinced that what can be achieved here with Reflex, which corresponds to the ideas exposed in [46], will meet the needs of the industry, giving the ability to computing entities to adapt themselves to the status of the external resources, and being able to do so in a flexible manner. This is of major relevance for the computing world of tomorrow,

where a lot of small-sized mobile nodes will be connected by a limited and fluctuating network.

7.3 Conclusions

In this dissertation we created a Java reflective system, Reflex, which allows a Java programmer to obtain reflective objects, *i.e.* objects that have an associated metaobject.

This reflective system provides transparent type-compatibility of reflective objects with their non-reflective equivalents, offers means to dynamically compose and decompose metaobjects associated to an instance, allows for conversion of normal objects to reflective objects, and sets up an open metaobject protocol that can be easily extended.

The Reflex system was developed with the objective in mind to offer flexibility in data space management mechanisms in mobile object systems. It has first been extended to be operational in distributed environments, and then has been put into concrete use.

Two kinds of metaobjects implementing two different resource management policies have been developed. The applicability of Reflex and those metaobjects has been tested in several examples of mobile agent programs.

Therefore we have achieved our objective to provide for more flexibility in mobile object systems, limiting ourselves to the aspect of data space management mechanisms as of now. However, when concluding this work, we have foreseen other applications of Reflex to offer more adaptability in different aspects of mobile object systems, such as behavior customization and adaptation to the network state.

Appendix A

Paradigms of Distributed Computing

This appendix aims at giving a definition of the different paradigms of distributed computing, in order to introduce that of mobile agents.

The focus is put on the mobile agent paradigm, highlighting its current drawbacks.

A.1 Client-Server

The client-server model is the well-known and widely used approach to distributed computing problems. In this model, the server entity offers a set of services that provide access to some of its local resources. This ability to interpret the demands of clients is called the know-how. The code that implements those services is local to the server's host, thereby the clients that require the execution of a certain service need to know how to invoke them. Servers are able to process the requests using their processor capability.

Hence in this paradigm, the server holds both the resources and the know-how, while the client component does not hold anything.

A.2 Remote Evaluation

In the remote evaluation paradigm a component has the know-how necessary to perform the service but it lacks the resources required, which are located at a remote site. Consequently, the component sends the service know-how to a computational component located on the remote site. The remote component executes the code using the resources available there and additionally delivers the result back.

A.3 Code on Demand

The general idea behind this model is that whenever one host is unable to perform a task due to lack of know-how, it must be possible to retrieve this knowledge from another entity to which the host is connected. When the host receives the needed information, it uses its own processor capability and resources to perform the desired task.

The code on demand model differs from the client-server model in a way that the notion of server (central host holding resources, know-how and processor capability) disappears, distributing the resources and processor capability through the hosts and letting the know-how be downloaded when not available.

Java applets and servlets are good examples of technologies implementing the code on demand paradigm. Applets are downloaded from the Web by hosts and execute locally, while servlets are transferred to remote Web servers with the purpose of executing there.

A.4 Mobile Agent

A.4.1 What's a Software Agent?

There is yet no consensus about the definition of what an agent is. From the end user perspective, it is said that an agent is essentially a program that is able to assist people and act on their behalf [43]. From this point of view we can say that a user can delegate tasks to the agent which is then responsible to perform them.

Pattie Maes gives in [48] a similar but slightly more specific definition. She views agents as "software that is proactive, personalized, and adapted. Software that can actually act on behalf of people, take initiative, make suggestions, and so on". She emphasizes the notion of proactivity of an agent, its ability to decide by itself or suggest, which goes a little beyond the first definition we can find in [43].

From a system perspective, an agent can be seen as a software object that resides in a certain execution environment. Danny Lange insists in [43] on the fact that what actually makes agents a major breakthrough is the fact that they can act asynchronously and autonomously inside their environment. They can be left executing continuously without needing any interaction of its creator or other entities involved. Additionally, agents are created to perform a predicted goal that will define its execution completely. During this execution they are sensible to changes in their environment, being able to react accordingly.

In addition to the common features mentioned above, agents may have other properties. An agent can be:

- communicative - it is able to communicate with other agents,

- mobile - it can travel from one host in the network to another,
- learning - it is able to adapt its behavior based on previous experiences.

A.4.2 The Paradigm

The mobile agent paradigm is a distributed computing model based on the notion of a software agent, defined as above, which has the extra property of mobility, and also - it is very often related to software agents in general and even more often to mobile agents - which is communicative.

A mobile agent is a software object that has the ability to travel through the hosts of a network. This means that when it is created its action is not restricted to the creation environment. It can move from one host to another where its work is needed, taking advantage of its mobility.

When an agent is transferred to the other host, it must carry his state and code in order to be able to resume execution in the different environment. Here, state can be defined by the agent's attributes that determine what to perform, while the code defines how to perform, when an agent arrives at the destination.

The mobile agent paradigm is said to be an improved mixing of the client-server and code on demand paradigms [43]. Here the client and the server are one single host, with its own resources and processor capability. The applet and servlet functions are now replaced by an emergent entity called mobile agent. This entity holds the necessary know-how to perform certain type of operations, and due to its mobility that knowledge becomes available throughout the network where the mobile agent operates.

A.4.3 Drawbacks of the Mobile Agent Paradigm

The mobile agent paradigm has not yet obtained the global acceptance from the industrial and research community. There are several reasons to this, either due to complications and weaknesses of the mobile agent approach, or due to the existence of well (or at least more) established alternatives.

The challenging difficulties of mobile agents systems

As mentioned in [27], security is a crucial issue when using mobile agents systems. In more traditional distributed computing approaches, security is more or less guaranteed nowadays. But when switching to the mobile agent paradigm, the entire architecture of distributed systems is changed. Thus security turns out to exist under many different forms that did not appear in previous paradigms. There are three aspects to security in mobile agents systems: protecting agents from malicious agents, hosts from malicious agents, and also agents from malicious hosts. Each of these issues has to be explored in depth before the mobile agents systems rise.

There are other challenging concerns when using mobile agents such as interoperability with other systems, coordination and communication aspects, and management of large societies of mobile agents [29]. Concerning interoperability with existing systems, the OMG recently proposed MASIF (Mobile Agent Systems Interoperability Facilities), a standard that deals with interoperability issues between different agent systems and CORBA services. Other approaches, such as the one exposed in [29], bet on the fact that mobile agents have to be merged with the existing WWW network, thus integrated in Web browsers and Web servers.

Mobile agents are not always well-suited

Another point adding to the difficult emergence of mobile agents is the fact that this approach is not always very well suited for distributed applications. In fact, moving agents over the network is good when the advantage of locality counterbalances the price of transferring more raw data. For instance, one can easily imagine that an email delivery system implemented with mobile agents will certainly work perfectly and be very close to the physical world analogy of mail delivery, but will be far less efficient than the actual approach using SMTP.

The competing alternatives to mobile agents

There are several existing and established alternatives to mobile agents that can achieve similar outcomes, as discussed in [52]. Among those alternatives are message passing systems, advanced forms of remote procedure call such as remote method invocation, or Common Object Request Broker Architecture (CORBA).

For instance, the Knowledge Query Manipulation Language (KQML) is one of the more advanced message passing system, which allows much more complex forms of interactions between agents than simple query/response mechanisms. With such advanced collaboration systems, agents need not move to hosts, they can just interact by passing message, through a simple transport mechanism.

CORBA is a platform and language independent mechanism for invoking remote object methods. CORBA can be used to create distributed systems that execute on many platforms, in many languages. CORBA is a direct threat to mobile agents because of its great portability and flexibility, and would allow developers to create agents that are capable of complex communication without ever traveling across a network.

A.5 Summary

In this appendix we have introduced the different paradigms of distributed computing, client-server, remote evaluation, code on demand and mobile agent.

We focused on the emerging mobile agent paradigm, highlighting its drawbacks and competing alternatives.

Appendix B

EzAgent

In order to perform the different tests needed to validate our work, we have developed a very primitive mobile agent platform for Java, EzAgent. This experimental platform based on RMI is primitive in the sense that it was not developed to be a *real* mobile agent platform, but just enough to perform the tests we needed.

For instance, the EzAgent platform does not provide for inter-agent communication mechanisms, since this was not needed for this thesis. Similarly, issues of security and performance were ignored. Though currently limited, this platform could be extended in the future to make it more operative.

B.1 Concepts

The concepts on which EzAgent is based are the same than any mobile agent platform, restricting ourselves to the existence and mobility of agents. An *agent* is an active object that spends its life in so-called *agent places*.

An agent life-cycle starts with its *creation* within a place. Then an agent can be *dispatched* to a remote place, either from its own will or from that of the owner or from the place itself. Possibly, it can be *deactivated*, that is made persistent on disk for a given deactivation time, and then *reactivated* to continue its activity. Finally, an agent ends its life by being *disposed*.

An agent place is responsible for agent-related administrative tasks, such as agent creation, reception, dispatching, deactivation and disposal. It is also supposed to offer a kind of yellow-pages service to contact an agent based on its identifier and to get the list of identifiers of the agents it is currently hosting.

B.2 The ezagent.EzAgent class

The `EzAgent` class is an abstract class that serves as the base class for any class of agents within the EzAgent platform. This class implements

method name	description
`void onCreation (Object[] args)`	invoked when a agent is created. This method should be used instead of the standard constructor.
`void onArrival()`	invoked each time an agent arrives to a new place.
`void onDispatch()`	invoked each time an agent is about to be dispatched to a remote place.
`void onDisposal()`	invoked when an agent is about to be disposed.
`void onDeactivation()`	invoked when an agent is about to be deactivated (written to disk for a time).
`void onActivation()`	invoked when an agent has just been reactivated (after deactivation time expired).

Table B.1: Customization methods of the life-cycle of an EzAgent.

the `java.lang.Runnable` interface, meaning it can be wrapped by a thread object that will execute it.

In order to allow programmatic access to the different life-stages of an agent, the `EzAgent` class defines methods that are automatically invoked and that can be redefined in subclasses (see table B.1). Only the `onCreation` method is declared abstract and therefore *must be* implemented by concrete subclasses, whereas the others are optional. This method replaces the normal constructor which should not be redefined in subclasses.

The dispatching, deactivation and disposal methods are implemented in the `EzAgent` class as *final* methods, since they should not be overridden. Recall that those final methods first invoke the corresponding life-cycle method to allow customization. Then they typically forward the request to the place where the agent presently resides.

Finally, an `EzAgent` keeps a reference to its current place, its home place, and has an internal state indicator (used to route the behavior each time the object is re-run), as well as an identifier.

B.3 The ezagent.EzPlace interface

The `EzPlace` interface defines the methods that should be implemented by a class representing a place for EzAgents (see table B.2). It defines methods for agent administration and yellow-pages services.

In the EzAgent platform, places are RMI remote objects that have a name used to look them up in the RMI registry.

method name	description
`void createAgent (String clsname, Object[] args)`	creates a new agent, instance of the class specified by `clsname`, using the specified arguments.
`void receiveAgent (EzAgent agent, String from)`	receives the specified agent sent by the place specified as `from`.
`void dispatchAgent (EzAgent agent, String to)`	dispatches the specified agent to the place specified by `to`.
`void dispatchAgent (EzAgent agent, EzPlace to)`	dispatches the specified agent to the specified place.
`void disposeAgent (EzAgent agent)`	disposes the specified agent.
`void deactivateAgent (EzAgent agent, long deact_time)`	deactivates the specified agent for the given time. The agent is stopped, and serialized to disk. It is reactivated after expiration of the deactivation time.
`Set getAllIds()`	returns a set containing the identifiers of all the agents hosted by the place.
`EzAgent getAgent (Integer agentId)`	returns a reference to the agent of the specified identifier.
`String getName()`	returns the name of this place.

Table B.2: The EzPlace interface.

B.4 The ezagent.EzPlaceImpl class

This class is our concrete implementation of the `EzPlace` interface. Such a place manages private hash tables that hold references to the active agents in the place, to the deactivated agents, and to the other known places.

In order to run an agent, such a place uses an `EzAgentRunner`, which is a thread object running the `Runnable` agent. When an agent creation request arrives, the place instantiates the agent and creates an agent runner for that agent. The runner simply runs the agent. When an agent is disposed or dispatched, the place throws an exception that causes the agent runner to stop (and therefore to be garbage collected later).

When a request for deactivation is received, the agent is serialized to disk, removed from the active agents hash table and added to the deactivated agents hash table, and a so-called `WakeUpThread` is created. The still running agent is then killed. The wake-up thread runs during the deactivation time of the agent, and when the time expires, it deserializes the agent and asks the place to reactivate it. Reactivating the agent implies re-putting it in the appropriate hash table, updating its internal state and creating a new agent runner.

B.5 Summary

Hereby we have quickly presented the EzAgent platform, its concepts and implementation guidelines.

A lot of work remains to be done to make a true agent platform from it: security issues and performance (including garbage collection) have to be dealt with.

Also, in order to make experiments of agents interaction, a mechanism for inter-agent communication should be designed and implemented on top of it.

In any case, this simple platform was sufficient for us to test our work, since it meets the necessary requirements we had: offering mobility and persistence features for agents, and being compatible with the JDK 1.2.

Appendix C

Resource Manager for Java

A resource manager, or resource repository, is a local Java object that is used to access local resources. It is basically an interface to a storage object where local resources are bound to resource identifiers. A local resource is any Java object that is specific to the running JVM and that offers services to incoming entities in the host. A resource manager is necessary to implement a rebinding policy (section 5.2).

As a matter of fact, Java does not provide a standard resource manager. We believe this would be useful, in particular when taking into account the fact that most mobile object systems today are based on this language. It is indeed in the area of mobile agent systems that such a resource manager finds all its usefulness.

In order to be able to test our metaobject for the rebinding policy, we implemented a simple resource manager for Java. We have extended the `java.lang.System` class in order to provide a way to get the local resource manager.

C.1 The java.lang.ResourceManager class

Our implementation of a resource manager is simple: a `ResourceManager` object aggregates a hash table of bindings resource identifier–resource reference. A resource identifier is simply a string identifying a resource.

The methods of a `ResourceManager` allow any object to bind a resource to an identifier, to unbind a resource given its identifier, and to obtain a reference to a resource based on its identifier.

The binding mechanism is based on that of the RMI registry: there are in fact two methods for binding a resource, `bind` and `rebind`. Using the `bind` method, if a resource is already bound to the specified string identifier, then a `ResourceAlreadyBoundException` is thrown. Using the `rebind` method, if a resource is already bound to the specified string identifier, then the binding is updated. Below is the code of these two methods:

```
/**
 * Binds the specified Object to the string id.
 * If a binding with that id already exists, then a
 * (runtime) ResourceAlreadyBoundException is thrown.
 */
public void bind(String name, Object value) {
    if(bindings.containsKey(name))
        throw new ResourceAlreadyBoundException(name);
    else
        bindings.put(name,value);
}

/**
 * Binds the specified Object to the string id.
 * As opposed to <code>bind</code>, this method
 * never throws exception. If a binding already exists,
 * then it is replaced by the new value.
 */
public void rebind(String name, Object value){
    bindings.put(name,value);
}
```

Unbinding a resource is simply done by removing the entry in the hash table:

```
/**
 * Unbinds the resource that is bound to the
 * specified identifier.
 * If no resource is bound to it, does nothing.
 */
public void unbind(String name){
    bindings.remove(name);
}
```

Finally, the `getResource` method allows any object to get a reference to a resource based on its identifier. If no resource is bound to the given identifier, then a `ResourceNotBoundException` is thrown. This exception is a run-time exception.

```
/**
 * Returns the resource (Object) bound to the given name.
 * If no resources are bound to this name, then a
 * (runtime) ResourceNotBoundException is thrown.
 */
public Object getResource(String name){
```

```
        if(bindings.containsKey(name))
            return bindings.get(name);
        throw new ResourceNotBoundException(name);
    }
```

A `ResourceManager` can automatically bind some default resources when initialized. This is specified by giving a boolean argument to the constructor. If this boolean is true, then the `defaultInit` method is invoked. Presently this method binds the standard input/output streams of a JVM:

```
    /**
     * Adds three default bindings:
     *  - "stdout" for System.out
     *  - "stderr" for System.err
     *  - "stdin" for System.in
     */
    protected void defaultInit(){
        bindings.put("stdout", System.out);
        bindings.put("stderr", System.err);
        bindings.put("stdin", System.in);
    }
```

C.2 The new java.lang.System class

In order to allow any object to get a reference to a unique local resource manager, we have extended the `System` class of Java.

Since we want a unique `ResourceManager` object per JVM, we have added a static instance variable to that class:

```
    private static ResourceManager resourceManager = null;
```

Then, the static method `getResourceManager` returns a reference to the manager, possibly initializing it if not yet created. A true argument is given to the constructor of the `ResourceManager` object in order to make it automatically bind the default resources (input/output streams).

```
    public static ResourceManager getResourceManager() {
        if(resourceManager == null)
            resourceManager = new ResourceManager(true);
        return resourceManager;
    }
```

This is all we need to setup a simple but yet operational mechanism of local resources binding/querying. Using our implementation, an object that wants to get a reference to the standard output will typically do the following:

```
    out = System.getResourceManager().getResource(``stdout'');
```

C.3 Summary

In this appendix, we have explained the necessity of a resource manager for Java and presented our simple implementation of it.

We also detailed the extension made to the `java.lang.System` class in order to make our resource manager unique and accessible in each running JVM.

This work was necessary for us to experiment truly with the rebinding policy.

Bibliography

[1] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *LNCS*, pages 111–130. Springer, April 1997.

[2] M. Ancona, W. Cazzola, G. Dodero, and V. Gianuzzi. Channel Reification: a Reflective Model for Distributed Computation. In *Proceedings of IEEE International Performance Computing, and Communication Conference (IPCCC'98)*, pages 32–36. IEEE, February 1998.

[3] M. Baldi, S. Gai, and G. P. Picco. Exploiting Code Mobility in Decentralized and Flexible Network Management. In *Mobile Agents: 1st International Workshop MA '97*, volume 1219 of *LNCS*. Springer, April 1997.

[4] L. Bergmans. The Composition Filters Object Model. Dept. of Computer Science, University of Twente, 1994.

[5] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas. An Architecture for Next Generation Middleware. In *Proceedings of Middleware'98*, pages 191–206. Springer-Verlag, September 1998.

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

[7] S. Bouchenak. Pickling Threads State in the Java System. In *Proceedings of the Third European Research Seminar on Advances in Distributed Systems (ERSADS'99)*, April 1999.

[8] S. Bouchenak and D. Hagimont. Approaches to Capturing Java Threads State. In *Proceedings of Middleware'2000*, April 2000.

[9] M. N. Bouraqadi-Saâdani. Un cadre réflexif pour la programmation par aspects. In *Langages et Modèles à Objets (LMO'99)*, Villefranche sur Mer - France, January 1999. Hermes.

[10] M. N. Bouraqadi-Sadani, T. Ledoux, and F. Rivard. Safe Metaclass Programming. In *Proceedings of OOPSLA'98*. ACM, October 1998.

[11] M. Braux and J. Noyé. Changement dynamique de comportement par composition de schémas de conception. In *Langages et Modèles à Objets (LMO'99)*, Villefranche sur Mer, France, January 1999.

[12] M. Braux and J. Noyé. Towards Partial Evaluating Reflection in Java. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Boston, MA, USA, January 2000. ACM Press. ACM SIGPLAN Notices, 34(11).

[13] J. P. Briot and P. Cointe. Programming with Explicit Metaclasses in SmallTalk-80. In *Proceedings of OOPSLA'89*, volume 24 of *Sigplan Notices*, pages 419–431. ACM, October 1989.

[14] T. Cai, P. Gloor, and S. Nog. DataFlow: A Workflow Management System on the Web using transportable Agents. Technical Report TR96-283, Dept. of Computer Science, Dartmouth College, Hanover, NH, 1996.

[15] D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in Java. In *Concurrency Practice and Experience*, volume 10. Wiley and Sons, Ltd., September 1998.

[16] A. Carzaniga, G. P. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, pages 22–32. ACM Press, 1997.

[17] W. Cazzola. Evaluation of Object-Oriented Reflective Models. In *Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98), 12th European Conference on Object-Oriented Programming (ECOOP'98)*, 1998.

[18] S. Chiba. Javassist Home Page. `http://www.hlla.is.tsukuba.ac.jp/~chiba/javassist/index.html`.

[19] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 30 of *SIGPLAN Notices*, pages 285–299. ACM, 1995.

[20] S. Chiba. Javassist — A Reflection-based Programming Wizard for Java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in Java and C++*, October 1998.

[21] S. Chiba. Load-time Structural Reflection in Java. To appear at ECOOP'2000, June 2000.

[22] S. Chiba and T. Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. In *Proc. of the 7th European Conference on Object-Oriented Programming*, volume 707 of *LNCS*, pages 482–501. Springer Verlag, 1993.

[23] G. Cohen, J.S. Chase, and D.L. Kaminsky. Automatic Program Transformation with JOIE. In *USENIX Annual Technical Conference '98*, 1998.

[24] P. Cointe. MetaClasses are first class objects: the ObjVLisp model. In *Proceedings of OOPSLA'87*, volume 22 of *Sigplan Notices*. ACM, October 1987.

[25] M. Dahm. Byte Code Engineering with the JavaClass API. Technical Report B-17-98, Institut für Informatik, Freie Universität Berlin, January 1998.

[26] J.-C. Fabre, V. Nicomette, T. Pérennou, R. J. Stroud, and Z. Wu. Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming. In *Proceedings of FTCS-25 "Silver Jubilee"*. ACM, June 1995.

[27] W.M. Farmer, J.D. Guttmann, and V. Swarup. Security for Mobile Agents: Issues and Requirements. In *Proceedings of NISSC'96*, 1996.

[28] J. Ferber. Computational Reflection in Class Based Object Oriented Languages. In *Proceedings of OOPSLA'89*, volume 24 of *Sigplan Notices*, pages 317–326. ACM, October 1989.

[29] S. Fnfrocken and F. Mattern. Mobile Agents as an Architectural Concept for Internet-based Distributed Applications – The WASP Project Approach. In *Proceedings of Kommunikation in Verteilten Systemen (KiVS'99)*, pages 32–43. Springer-Verlag, 1999.

[30] G. H. Forman and J. Zahorjan. The Challenges of Mobile Computing. In *IEEE Computer*, volume 27, pages 38–47, 1994.

[31] M. Fowler. *UML Distilled*. Object Technology Series. Addison-Wesley, 1997.

[32] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. In *IEEE Transactions on Software Engineering*, volume 24, 1998.

[33] S. Fünfrocken. Transparent Migration of Java-based Mobile Agents (Capturing and Reestablishing the State of Java Programs. In *Proceedings of the Second International Workshop on Mobile Agents (MA'98)*, volume 1477 of *LNCS*, pages 26–37. Springer-Verlag, September 1998.

[34] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series, 1995.

[35] M. Golm and J. Kleinöder. Jumping to the Meta Level, Behavioral Reflection Can Be Fast and Flexible. In *Proceedings of Reflection '99*, volume 1616 of *LNCS*, pages 22–39. Springer Verlag, 1999.

[36] Object Management Group. CORBA: Architecture and Specification, August 1994.

[37] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C.V. Lopes, C. Maeda, and A. Mendhekar. Aspect Oriented Programming. In *Special Issues in Object-Oriented Programming*. Max Muehlhaeuser (general editor) et al., 1996.

[38] G. Kiczales, J. Des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[39] J. Kleinöder and M. Golm. MetaJava: An Efficient Run-Time Meta Architecture for Java. In *Proceedings of the International Workshop on Object Orientation in Operating Systems (IWOOS'96)*. IEEE, 1996.

[40] P. Knudsen. Comparing two Distributed Computing Paradigms - a Performance Case Study. Master's thesis, University of Tromsø, 1995.

[41] J. Lamping, G. Kiczales, L. Rodriguez, and E. Ruf. An Architecture for an Open Compiler. In *Proceedings of the International Workshop on Reflection and Meta-Level Architectures*, pages 95–106. Akinori Yonezawa and Brian C. Smith, editors, 1992.

[42] D.B. Lange. Java Aglets Application Programming Interface (J-AAPI). IBM Corp. White Paper, February 1997.

[43] D.B. Lange. Mobile Objects and Mobile Agents: The Future of Distributed Computing? In *Proceedings of ECOOP'98*, July 1998.

[44] D.B. Lange and D.T. Chang. IBM Aglets Workbench — Programming Mobile Agents in Java. IBM Corp. White Paper, September 1996.

[45] T. Ledoux. OpenCorba: a Reflective Open Broker. In *Reflection'99*, volume 1616 of *LNCS*. Springer Verlag, 1999.

[46] T. Ledoux and M. N. Bouraqadi-Saâdani. Adaptability in Mobile Agent Systems using Reflection. RM'2000, Workshop on Reflective Middleware, http://www.comp.lancs.ac.uk/computing/rm2000/, April 2000.

[47] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87*, pages 147–155. ACM Sigplan Notices, 1987.

[48] P. Maes. Pattie Maes on Software Agents. In *IEEE Internet Computing*, July 1997.

[49] J. McAffer. Meta-Level Programming with CodA. In *Proceedings of the 9th Conference on Object-Oriented Conference (ECOOP'95)*, volume 952 of *LNCS*, pages 190–214. Springer-Verlag, 1995.

[50] P. Mulet, J. Malenfant, and P. Cointe. Towards a methodology for explicit composition of metaobjects. In *Proceedings of OOPSLA'95*, pages 316–330. ACM Sigplan Notices, October 1995.

[51] R. Rao. Implementational Reflection in Silica. In *Proceedings of ECOOP'91*, pages 251–266. Springer-Verlag, July 1991.

[52] D. Reilly. Mobile Agents – Process migration and its implications. `http://www.davidreilly.com/topics/software_agents/`, 1998.

[53] T. Sekiguchi, H. Masuhara, and A. Yonezawa. A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation. In *Proceedings of Coordination'99*, LNCS. Springer-Verlag, 1999.

[54] B. C. Smith. Reflection and Semantics in Lisp. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, January 1984.

[55] M. Straßer, J. Baumann, and F. Hohl. Mole—A Java Based Mobile Agent System. In M. Mühlaüser, editor, *Special Issues in Object-Oriented Programming: Workshop Reader of the 10th European Conference on Object-Oriented Programming ECOOP'96*, pages 327–334. dpunkt, July 1996.

[56] Sun Microsystems, Inc. The Java Language Specification, 1996.

[57] Sun Microsystems, Inc. Java Object Serialization Specification – JDK1.2, November 1998.

[58] Sun Microsystems, Inc. Object Serialization. `http://java.sun.com/products/jdk/1.2/docs/guide/serialization/`, 1998.

[59] Sun Microsystems, Inc. Remote Method Invocation. `http://java.sun.com/products/jdk/1.2/docs/guide/rmi/`, 1998.

[60] Sun Microsystems, Inc. Dynamic Proxy Classes. `http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html`, 1999.

[61] Sun Microsystems, Inc. Reflection API Documentation. `http://java.sun.com/products/jdk/1.2/docs/guide/reflection/`, 1999.

[62] M. Tatsubori. An Extension Mechanism for the Java Language. Master's thesis, University of Tsukuba, Japan, 1999.

[63] A. R. Tripathi, N. M. Karnik, R. D. Singh, T. Ahmed, J. Eberhard, and A. Prakash. Development of Mobile Agent Applications with Ajanta. Technical report, Department of Computer Science, University of Minnesota, Minneapolis, 1999.

[64] G. Vigna. Mobile Agents and Security. In *LNCS State-of-the-Art Survey*. Springer, 1998.

[65] I. Welch and R. Stroud. From Dalang to Kava — The Evolution of a Reflective Java Extension. In *Proceedings of Reflection '99*, volume 1616 of *LNCS*, pages 2–21. Springer Verlag, 1999.

[66] J. E. White. Telescript technology: The Foundation for the Electronic Marketplace. White paper, General Magic, Inc., 1994.

[67] Z. Wu. Reflective Java and A Reflective-Component-Based Transaction Architecture. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*. J.-C. Fabre and S. Chiba, eds, 1998.

[68] Y. Yemini. The OSI Network Management Model. In *IEEE Communications*, pages 20–29, May 1993.

[69] Y. Yokote. The ApertOS Reflective Operating System: The Concept and Its Implementation. In *Proceedings of OOPSLA'92*, volume 27 of *Sigplan Notices*, pages 414–434. ACM, October 1992.