Vrije Universiteit Brussel - Belgium

Faculty of Sciences

In Collaboration with Ecole des Mines de Nantes - France

and

Universidad de Chile - Chile

2006

A Metaobject Protocol for AmbientTalk based on Mirror Methods

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: Dieter Standaert

Promoter: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promoter: Éric Tanter (Universidad de Chile)

Abstract

Ambient-Oriented Programming (AmOP) languages are especially designed for software development for pervasive and ambient computing. In this context, reflective abilities are highly desired to be able to create adaptive software. In this thesis, we propose a reflective architecture for ambient actors, instantiated in the AmOP language AmbientTalk. Our architecture is structured according to different levels of abstraction, distinguishing between the metalevel for regular objects and active objects. The architecture adopts at its core the concept of mirrors and mirror methods to safeguard object encapsulation even in the presence of powerful reflective facilities, such as access to the VM.

"The ability to use a mirror to introspect parts of oneself is a striking example of evolutionary convergence."

- Diana Reiss

Acknowledgments

I would like to thank my promotors prof. Dr. Theo D'Hondt and prof. Dr. Éric Tanter for giving me the opportunity to do this research. I would like to give my utmost gratitude to prof. Dr. Éric Tanter for his continuous support and encouraging spirit throughout the development of this thesis. Even in the most hectic days he gifted me with a positive stance towards the work. Special thanks to prof. Dr. Theo D'Hondt for referring me to the EMOOSE program for which I submitted this thesis. It has proven to be a great life experience for which I am truly grateful. Without a doubt it has brought me lasting memories and taught me a great deal about myself. I guess I did a lot of reflection this year.

I want to show my gratitude and respect to the people whom organized the EMOOSE program. It has truly been a privilege to be taught by professors from different countries all around the world and to be confronted with such a variety in culture. A happy greet and cheerful thank to my colleages Benoit, Sergio and Angel from EMOOSE. We had a great year together and I have learned a great deal about a great many things from you guys! I hope my teachings in beer-ology returned the favor. Many thanks to the people which have lived with me and endured me during my stay in Chile. Marisol, Jane, Barbara and Benoit have shown me a great time, even during the stressful periods of my research. A lot of gratitude towards my friends at home, whom kept close to me despite the great distance. In particular a big thanks to David and Mehdi for their endless patience with his far away friend. I wish to thank my parents for their enormous support and encouragements which they gave me zealously. And finally I would like to thank my girlfriend who proved to be my greatest supporter in this peculiar period.

Contents

I Introduction And State Of The Art **1**

1 Introduction **2**
1.1 Ambient intelligence . . . 2
1.2 Reflection . . . 3
1.3 Problem statement and objective . . . 4
1.4 Overview of the thesis . . . 5

2 Previous Research **7**
2.1 AmbientTalk . . . 7
2.1.1 Passive object layer . . . 8
2.1.2 Active object layer . . . 10
2.2 Extreme encapsulation . . . 12
2.2.1 Conflict by extension . . . 13
2.2.2 Conflict by cloning . . . 14
2.2.3 Conflict by reflection . . . 16
2.2.4 Conclusion . . . 16
2.3 Mirrors . . . 17
2.3.1 A mirror-based reflective API . . . 17
2.3.2 Encapsulation . . . 19
2.3.3 Stratification . . . 20
2.3.4 Ontological correspondence . . . 20
2.4 Mirror methods . . . 20
2.4.1 The meta pseudo-variable . . . 21
2.4.2 The mirror methods . . . 21
2.4.3 A structural reflective API . . . 22
2.4.4 A behavioral reflective API . . . 24
2.4.5 Concluding remarks . . . 26
2.5 Summary . . . 27

II Passive Mirrors **28**

3 Design Of Passive Mirrors **29**

3.1 Passive mirror methods . . . 29
3.1.1 Design of passive mirror methods . . . 30
3.1.2 Mirror as native function . . . 31
3.2 Passive mirrors . . . 31
3.2.1 The metaobject . . . 32
3.2.2 Meta as pseudo-variable . . . 32
3.2.3 Design of passive mirrors . . . 33
3.2.4 Mirrors on mirrors . . . 33
3.3 Structural reflection . . . 34
3.3.1 Field mirrors . . . 34
3.3.2 Method mirrors . . . 35
3.3.3 Permissions for mirrors . . . 36
3.4 Behavioral reflection . . . 37
3.4.1 Mirrors as listeners . . . 37
3.4.2 Behavioral reflection on field accesses . . . 38
3.4.3 Behavioral reflection on method invocations . . . 38
3.5 Summary . . . 39

4 The AmbientTalk Interpreter 40
4.1 The main hierarchies . . . 40
4.2 The AbstractGrammar hierarchy . . . 41
4.3 The AGValue hierarchy . . . 41
4.4 Function implementation . . . 42
4.5 Continuation frames . . . 42
4.6 Native functions and thunks . . . 43
4.7 The Read-Eval-Print loop . . . 43
4.8 Example: declaring and evaluating a cloning method . . . 44
4.9 Summary . . . 46

5 Implementation Of Passive Mirrors 48
5.1 The native function mirror . . . 48
5.2 Implementation of passive mirror methods . . . 49
5.3 Implementation of meta as pseudo-variable . . . 51
5.4 The family of metaobjects . . . 52
5.4.1 Parents and their children . . . 52
5.4.2 The meta of objects . . . 53
5.4.3 The meta of field accesses . . . 54
5.4.4 The meta of method invocations . . . 55
5.4.5 The meta of fields . . . 55
5.4.6 The meta of methods . . . 55
5.4.7 The meta of field listeners . . . 56
5.4.8 The meta of method listeners . . . 56
5.5 Filtered searching . . . 56

5.6 The family of passive mirrors . . . 57
5.6.1 Passive mirrors for objects . . . 57
5.6.2 Field mirrors . . . 58
5.6.3 Method mirrors . . . 58
5.6.4 Listening mirrors for fields . . . 58
5.6.5 Listening mirrors for methods . . . 59
5.7 Summary . . . 59

III Active Mirrors 60

6 Design Of Active Mirrors 61
6.1 Reflection in the active layer . . . 61
6.1.1 Behavior . . . 62
6.1.2 Mailboxes . . . 62
6.1.3 Thread . . . 63
6.2 Active mirror methods . . . 63
6.2.1 Design of active mirror methods . . . 64
6.2.2 The native function amirror . . . 64
6.3 Active mirrors . . . 65
6.3.1 The metaobject of actors . . . 65
6.3.2 The metaActor pseudo-variable . . . 67
6.3.3 Design of active mirrors . . . 67
6.4 Structural reflection for actors . . . 67
6.4.1 Behavior mirrors . . . 68
6.4.2 Mailbox mirrors . . . 68
6.4.3 Message mirrors . . . 69
6.4.4 Thread mirror . . . 69
6.5 Behavioral reflection for actors . . . 70
6.5.1 Behavioral reflection on mailboxes . . . 70
6.6 Summary . . . 70

7 Implementation Of Active Mirrors 72
7.1 The native function amirror . . . 72
7.2 Implementation of the active mirror methods . . . 73
7.3 Implementation of metaActor as pseudo-variable . . . 74
7.4 The familiy of metaobjects for actors . . . 75
7.4.1 The meta of an actor . . . 75
7.4.2 The meta of mailboxes . . . 76
7.4.3 The meta of threads . . . 76
7.5 The familiy of mirrors for actors . . . 76
7.6 Summary . . . 76

IV Conclusions And Future Work 78

8 Evaluation Of The MOP 79
8.1 Reflection in the passive layer 79
8.2 Reflection in the active layer 80
8.3 A remote debugger 80

9 Conclusions And Future Work 82
9.1 Conclusions 82
9.2 Future work 84
9.2.1 More reflection on AmbientTalk 84
9.2.2 A reflective virtual machine 85

List of Figures

2.1 The double-layered object model of AmbientTalk. Passive objects implement both the behavior and mailboxes of the active objects. 8
2.2 In AmbientTalk passive objects can be created ex-nihilo, by extending or cloning an existing object. Cloned objects share the parent of the original object. 9
2.3 At the level of the interpreter, passive objects are represented by a list of variables and a list of constants. 9
2.4 A message is sent from the `outbox` of to sending actor to the `inbox` of the receiving actor. Upon successful delivery, the sender moves the message in the `sentbox` mailbox. Meanwhile the receiver may start processing the message. When the message is evaluated, the receiver moves the message from `inbox` to `rcvbox`. 12
2.5 In prototype-based languages, objects can be dynamically extended and child objects gain access to the private state of their parent. This is a breach of encapsulation. 14
2.6 Supporting only a default cloning operation in a prototype object forces the inclusion of setters for the internal state of an object so that it may be initialized properly. This can be a breach of encapsulation of an object's private state. 15
2.7 In Self, reflection on an object is done through mirrors. Such a mirror can offer access to individual slots by returning a mirror to that slot. 17
2.8 In class-based object-oriented languages, the class represents the metalevel of an object and is usually accessible through a message send to the object. 18
2.9 Mirrors offer metafacilities on an object, its class and superclass. 19
2.10 Prototype point object featuring a mirror method. 22
2.11 Mirror on a point object. 22

3.1 A mirror methods is a decorator of a method in AmbientTalk. 31
3.2 A conceptual schema of passive mirrors. 32
3.3 The relationship between metaobjects and their shared metafunctions. . . . 33
3.4 A UML representation of the relation between objects, metaobjects and mirrors. 34
3.5 A mirror on a mirror. 34

3.6 UML diagram of a mirror on a mirror. 35

4.1 The parse-tree returned from parsing a cloning method. 44
4.2 The initial steps for evaluating the declaration of a cloning method. 45
4.3 The evaluation of the *cloning* attribute in the declaration. 45
4.4 The sequence of evaluating the cloning method. 46

5.1 Passive mirror methods decorate a normal function in AmbientTalk. 50

6.1 An active mirror method wraps a normal method and extends only the behavior of the method application. Active mirror methods follow the decorator pattern for methods. 65
6.2 The different layers of AmbientTalk objects and their mirrors. 65
6.3 The relation between an active mirror and its creator. 67
6.4 The metaobject of an actor is stored locally. 67
6.5 The meta actor is an environment wrapped around the base actor. 68

7.1 Active mirror methods are decorators of a normal AmbientTalk function. . 74

9.1 Exposing VM facilities with VM actors residing at the active object layer. . 86

Part I

Introduction And State Of The Art

Chapter 1

Introduction

We begin this chapter with an introduction on the domain of this thesis, namely Ambient Intelligence in section 1.1. We continue with a discussion on reflection in section 1.2. In section 1.3 we state the problem discussed in this thesis and present the objectives of our research. Finally we give an overview of the thesis in section 1.4.

1.1 Ambient intelligence

The modern evolution of computing hardware is characterized by an increase in computing power and a decrease in hardware scale. In unison with the miniaturization of hardware came the development of mobile hardware. Mobile devices such as portable computers, smartphones and PDA's progress in computing power, storage capacity, autonomy and connectivity. While early mobile hardware such as PDA's relied on hardwiring for synchronization with a central agenda, new models increasingly incorporate wireless technologies such as WiFi and Bluetooth. The growth of wireless communication between mobile devices gave birth to *mobile networks*. Mobile networks surround mobile devices equipped with wireless technology and enable them to communicate with other devices in their environment. This form of pervasive computing is called Ambient Intelligence, named by the European Council's IST Advisory Group [15].

Mobile networks surrounding mobile devices have characteristics that distinguish them from normal networks. Connections are volatile because the communication range of wireless technology is limited and the boundaries of the network change as the mobile device moves. The unheralded emergence and quietus of networks necessitates the use of open networks. These properties encumber software developers to cope with ad hoc networks and security issues. As ambient hardware represents an individual computing unit, applications need a high level of autonomy and have a natural concurrency. Current industrial programming languages such as Smalltalk [12], Java [14] offer no inherent support for this kind of pervasive computing. They rely on low-level system software and libraries such as JXTA [13] or M2MI [16] but development of software for ambient hardware still remains difficult.

Within this context of pervasive computing, De Meuter introduces ChitChat [4]. De Meuter claims that current mainstream programming languages are inadequate for programming applications in ambient environments. With ChitChat, a delegation-based, prototype-based language, he shows that dynamic, self-sufficient objects are more suitable for programming in ambient environments. To tackle security issues, inherent from the openness of ambient networks, he proposes a fundamental language principle to ensure safety at the level of the language named *extreme encapsulation*. In [6], De Meuter et al. show that *method attributes* as a language feature provide flexible object *extension*, *cloning* and *reflection* without compromising *extreme encapsulation*.

Meanwhile, Dedecker and Van Belle, propose another direction to facilitate programming in ambient environments in [8]. The reason that programming concurrent, distributed software in open distributed environments is best accomplished using actors, based on the actor model of Agha [1]. But ambient environments stress the actor model from Agha in that the environments are *highly dynamic* and *less reliable*. In [8] Dedecker and Van Belle propose to extend the Agha's actor model to what they name *the ambient actor model*. They add semantics for explicit message sending and mailboxes management. Using mailboxes they add the notion of *pattern based communication*, a contextual identification mechanism for actors. The ambient actor model led to a new programming paradigm named the *Ambient-Oriented Programming Paradigm* [7] (or AmOP) which consists of programming languages that incorporate support for the strains of an ambient environment at the level of the language. Within the context of AmOP, Dedecker et al. present AmbientTalk in [9] a novel language designed to tackle the characteristics of ambient environments at the level of the language.

1.2 Reflection

The beginnings of reflection go back to the days where von Neumann introduced a computer architecture where program instructions are stored alongside with other computational data on which programs operated. In [30] von Neuman clearly states his fascination for programs manipulating other programs. However, at that point in time, the distinction between a computational system, a metasystem and a reflective system were not clearly defined. Later, the distinction between a program and a computational system is further clarified by Steyaert in [25]: a program is a representation of the computational system. To be useful for reflection, the representation must be up-to-date with respect to the domain and operations on the representation should be effective in the computational system. The concept of reflection as we know it today was first introduced by Pattie Maes in [19]. Pattie Maes defines a reflective architecture as,

> *"A programming language is said to have a* reflective architecture *if it recognizes reflection as a fundamental programming concept and thus provides tools for handling reflective computation explicitly."*

In [10] Smith mentions two important requirements for a language to be called reflective. First, the language needs a description of the language from within itself. Second, this self-representation must be causally connected to the system. These two requirements translated themselves to structural and behavioral reflection. The existence of an interface for reflection then led to the concept of *metaobject protocols*, abbreviated as MOP. Kiczales et al. define them as,

> *"Metaobject protocols are interfaces to the language that give users the ability to incrementally modify the language's behavior and implementation, as well as the ability to write programs within the language."* [17]

Meanwhile reflection found its place in programming languages such as 3-Lisp [22], Agora [25, 3], Smalltalk [21], CLOS [17].

Within the context of distributed computing, reflection has been intensively researched as mechanism to write flexible applications in a highly heterogeneous environments. In [4] De Meuter criticizes the use of reflection in open networks as dangerous. He distinguishes between reflection performed on an object using external operators and reflection done through the object using reflective methods. He considers the first category of reflection as harmful for an object's private state since the object has no control over which parts are reified at its metalevel and how those parts are exposed publicly. He states the principle of *extreme encapsulation* and shows that a language must uphold this principle in order to be considered safe for open networks [4]. Later, in the context of ambient-oriented programming, AmbientTalk was introduced [7, 9]. AmbientTalk included a small, but effective reflective mop to operate on its most notable language constructs. However, the metaobject protocol they introduced was not in accordance with *extreme encapsulation*. Even in later designs, extreme encapsulation gave way to flexibility [29]. In a recent extension of the reflective kernel from AmbientTalk, by Martin [20] , the principle of extreme encapsulation is not reinstated. In this thesis we state that *extreme encapsulation* is a paramount principle of ambient oriented programming and it should be a fundamental property of a reflective API. We now progress to the problem statement and objective of this thesis.

1.3 Problem statement and objective

AmbientTalk [9] is a novel language for the Ambient-Oriented paradigm [7]. It incorporates support for many characteristics of ambient environment at the level of the language. It also includes a small reflective frame [9], a valuable asset for applications in an ambient environment. However, the current reflective API offered by AmbientTalk is not in league with the extreme encapsulation principle proposed by De Meuter [4]. We believe that the extreme encapsulation principle as demonstrated in ChitChat [4] is a paramount language principle of the Ambient-Oriented paradigm [7] to solve security problems inherent from

the language. This contrasts with AmbientTalk's design to develop programs for Ambient Intelligence: its reflective framework is a potential security risk, a dangerous property in an environment based on *open networks*. Furthermore, its current reflective framework offers but a limited set of operations, used for reflection in the *active layer* of AmbientTalk.

The goal of this thesis is to propose and implement a design for a reflective API for AmbientTalk that supports *extreme encapsulation*. In [2] Bracha and Ungar have shown a design technique for reflective API's called *mirrors*. While *mirrors* offer solutions to many desiderata of a reflective framework, it is in violation with extreme encapsulation. In [27] Tanter shows how to reconcile extreme encapsulation in the design of mirrors using *mirror methods* for ChitChat. We take his interpretation of mirrors as an example and translate the concepts proposed by Tanter to AmbientTalk. In this way we hope to create a new reflective API for AmbientTalk that reinstates extreme encapsulation.

1.4 Overview of the thesis

In this thesis we present a design for a reflective API for AmbientTalk. We start with an overview of previous research on which the concepts of our design are based in chapter 2. We give a short introduction to AmbientTalk, the target language of our design, in section 2.1. Then we elaborate on *extreme encapsulation* and which typical language constructs tend to violate it in section 2.2. We continue our discussion with *mirrors* in section 2.3 and explain the benefits of reflective API's based on mirrors. In section 2.4 we show how mirrors are not in accordance with extreme encapsulation and explain how *mirror methods* adapt mirrors to be in league with extreme encapsulation. We end the chapter with a discussion on the benefits of a reflective API based on mirror methods.

Having explained the fundamental concepts which we try to uphold in our design, we can concretize the design in chapter 3 for the *passive object layer* of AmbientTalk. We explain how mirror methods serve as constructors for mirrors and expatiate on the elements involved in a mirror (and mirror method) based design. In chapter 4 we give a technical overview of the internal mechanics of the AmbientTalk interpreter. We identify the main classes and class hierarchies responsible for evaluating AmbientTalk expressions. We demonstrate what we have learned by examining the evaluation of a *cloning method* in detail.

After revising the technicalities of the AmbientTalk interpreter, we move on to chapter 5 where we implement the design proposed in chapter 3. We explain the design techniques adopted and problems encountered.

In chapter 6 we reintroduce the concepts of mirrors and mirror methods, but for the *active object layer* of AmbientTalk. We list the elements of actors we reflect on and design *active mirrors* and *active mirror mehods*. We implement the design in chapter 7 and elaborate on implementation choices.

We end our research with an evaluation of the reflective API proposed in this thesis. We demonstrate our implementation first, in chapter 8. Then we discuss the benefits and

downsides from the acquired metaobject protocol in AmbientTalk in chapter 9. In that same chapter, we also shed light on missing elements of our implementation and possible extensions that could benefit from the design proposed in this thesis.

Chapter 2

Previous Research

We give an overview of previous research which has led to the development of AmbientTalk and the formulation of the principles on which the metaobject protocol proposed in this thesis is founded. We begin with an introduction to AmbientTalk, the language in which we deploy our reflective API, in section 2.1. Then we examine the paramount *extreme encapsulation principle* [4, 6] our protocol is subject to in section 2.2. We move on to the design technique of mirrors in section 2.3, the design technique we applied for our proposed protocol. Finally, in section 2.4, we explore a proposed design for mirror methods [27], in ChitChat [4] which we will transmute to AmbientTalk to reflect on passive objects in chapter 3.

2.1 AmbientTalk

AmbientTalk [9] is a prototype-based, object-oriented programming language designed for the field of ubiquitous computing, named Ambient Intelligence by the European Council's IST Advisory Group [15]. This emerging field of ambient-oriented programming has led to the Ambient-Oriented Programming Paradigm [7]. It focusses on languages for software development in the context of mobile devices and their networks. Such mobile networks consist of a multitude of volatile devices connected through open networks. This poses extra strains on software developers to cope with inherent problems of ad hoc networks such as security issues and network failures. AmbientTalk is designed to aid software development in the context of mobile (ad hoc) networks. Another characteristic is that it follows the guidelines for object encapsulation for ambient environments [4, 6] on which we elaborate further in section 2.2. AmbientTalk is also the target language for the reflective framework proposed in this thesis. We elaborate on the language's feature characteristics in more detail.

One of AmbientTalk's most prominent features is its double-layered object model, depicted in figure 2.1. It discriminates between passive, or normal objects, and active objects which fulfill the role of actors in the network [7]. We discuss each layer in more detail.

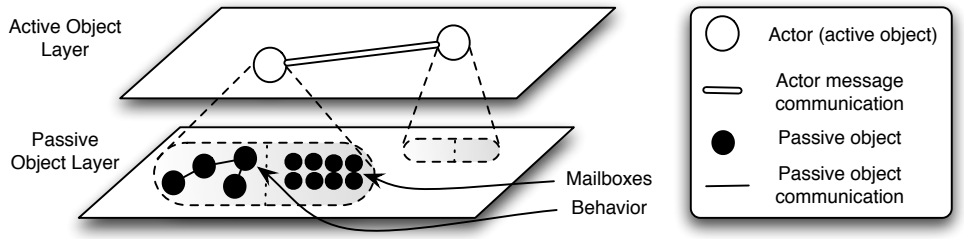

Figure 2.1: The double-layered object model of AmbientTalk. Passive objects implement both the behavior and mailboxes of the active objects.

2.1.1 Passive object layer

Passive objects in AmbientTalk are regular prototype-based objects. They can be created ex-nihilo, by extending other objects or by cloning existing objects using the following syntax,

```
anObject:object({
 ...
 cloning.makeClone()::{this};
 ...
});
extendedObject:extend(anObject,{
 ...
 aMethod()::{...};
 ...
});
aClone: extendedObject.makeClone();
```

A representation of these objects in given in figure 2.2. Extending an object results in a new object, which we name the child. The child keeps a reference to the original object, which we name its parent. When a field or method is looked up in the child, but not found, the interpreter searches the field or method in its parent object. This is the default technique for prototype-based objects to implement code-sharing [24]. Cloning an object returns an identical copy of the object, including its parent reference. In figure 2.2 we see that the clone, `aClone`, of `extendedObject` shares the same parent as `extendedObject`. In AmbientTalk, passive objects can send and receive messages, using call-by-reference. Following the prototype oriented paradigm, passive objects are entirely self-sufficient [24], an important feature of objects in an ambient environment [7].

At the level of the interpreter, passive objects in AmbientTalk are a collection of bindings where each binding connects a *name* to a *value*. Furthermore, objects have two distinct repositories to store bindings, one for variables and one for constants. Variables, as the name says, can change over time while constants have a fixed value. In AmbientTalk,

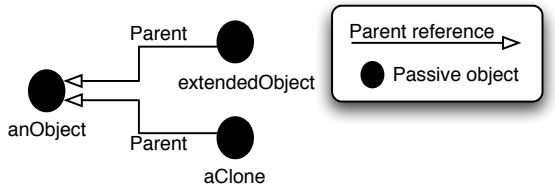

Figure 2.2: In AmbientTalk passive objects can be created ex-nihilo, by extending or cloning an existing object. Cloned objects share the parent of the original object.

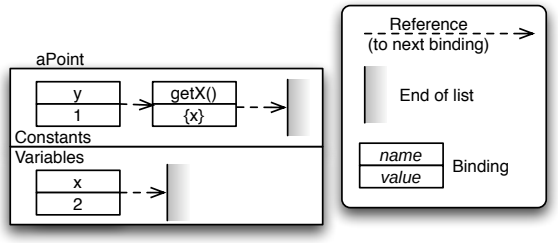

Figure 2.3: At the level of the interpreter, passive objects are represented by a list of variables and a list of constants.

we can declare a variable using the :-operator, while constants are declared with the ::-operator. The immutable nature of constants, makes them ideal candidates for public access. We demonstrate this with the following code,

```
aPoint:object({
 x:2; //variable x
 y::1; //constant y
 getX()::{x}; // 'constant' function returning x
});
aPoint.x; // --> error, did not find "x"
aPoint.y; // --> returns "1"
aPoint.y := 3 // --> error, "y" is immutable
aPoint.getX() // --> returns "2"
```

We create a point with a variable `x`-coordinate, a constant `y`-coordinate and a constant method which returns the value of `x`. An representation of this object at the level of the interpreter is given in figure 2.3[1]. The code shows that we cannot access `x` from outside the object. To obtain the value of `x`, we can use the method `getX`, which is declared as a constant and therefor publicly available. We can access the constant `y`, but we can not modify it. We conclude that modifying the internal state of an object can only be done through message passing, that is, by implementing a *public* method which modifies the *private, variable* state of the object. The repository of constants represents the object's public interface. An extended object has access to its parents internal state. The following code shows this,

```
aPoint:object({ ... as before ... });
e:extend(aPoint,{
 setX(val)::{x:=val}
})
```

[1]Note in figure 2.3 that each binding has a reference to the next binding; the collection of bindings is implemented as a list.

```
e.setX(3)
aPoint.getX() // --> returns "3"
```

We see that an extended object *can* modify its parent's private state.

2.1.2 Active object layer

The second object layer of AmbientTalk contains *only* active objects. Active objects in AmbientTalk follow the ambient actor model presented by Dedecker and Van Belle in [8]. It approximates the actor model by Agha in [1] closely, but unlike normal actors, active objects can be stateful. An actor or active object, is made up of a collection of passive objects. More specifically, an actor has one unique passive object describing its interface, called its *behavior*, which processes incoming messages of the actor. Each actor occupies a unique thread in the interpreter. Inside the actor, all interactions between its passive objects are executed sequentially within the actor's thread. Actors communicate asynchronously to integrate network delays, failures and thread synchronization. AmbientTalk provides asynchronous communication primitives between active objects using the `#`-operator as follows,

```
anActor#message(arguments)
```

Messages between actors may contain passive objects as arguments, which are passed by copy in accordance with the containment principle [6]. The containment principle states that a passive object can belong to only one actor. Actors can be created with the object that represents their behavior as shown below,

```
anActor:actor(object({
 show()::{display("Hello");}
 ... definition of the actor's behavior ...
};
anActor#show(); //--> "Hello"
```

The interaction between actors is characterized by a set of *mailboxes*. The actor holds four mailboxes, `inbox`, `rcvbox`, `outbox` and `sentbox`, which contain the messages received, processed, sent and delivered respectively. Incoming messages are taken from the `inbox`, processed in the behavior, then put in the `rcvbox`. Messages to other actors are put in `outbox` and moved to `sentbox` when the message was successfully delivered. AmbientTalk's virtual machine is responsible for sending and delivering the messages. We demonstrate how actors communicate based on the following code, and corresponding figure 2.4,

```
// actor1
actor1:actor(object({
 sendAMessageTo(anActor)::{
  anActor#aMethod(...args...)}};
// actor2
```

```
actor2:actor(object({
 aMethod(...args...)::{
  ... process message ...}}));
 // make actor1 send a message to actor2
 actor1#sendAMessageTo(actor2);
```

In the first phase `actor1` wants to send a message to `actor2`. It places the message in `outbox`. The interpreter takes the message from `outbox` (`actor1`) and sends it to `actor2`. Any message an actor receives is placed in `inbox`, so the interpreter places the received message from `actor1` in the `inbox` from `actor2`. If the message was successfully received, phase two begins.

In phase two, there may be a time discrepancy between `actor1` and `actor2` since both operate independently in a separate thread. `actor1` consideres the message send as completed and moves the message to `sentbox`. `actor2` removes the message from its `inbox` and places it on its evaluation stack to process the message. When the message is fully evaluated, `actor2` places the message in `rcvbox` and we continue to phase three.

In phase three, the message send was successfully processed on both sides. Note that both actors, `actor1` and `actor2`, have a copy of the message in their mailboxes, `sentbox` and `rcvbox` respectively. Actor embody exactly one thread and race conditions apply at the `inbox` access. This is solved by the interpreter.

Non-blocking communication between actors makes synchronization between actors difficult. However, synchronization may be be required between collaborating parties to ensure a consistent state. By storing the complete communication history in the mailboxes, actor's possess a reified communication tree. When an inconsistent state is detected, they can navigate backwards over the reified communication tree to recover to a previous, consistent state.

In addition to the four mailboxes for communication, active objects keep four other mailboxes for ambient service discovery. An actor can offer services by adding a *pattern* in a mailbox named `providedbox`. The contents of this mailbox describes the services an actor provides and the interpreter is responsible for broadcasting this service to the ambient environment. Additionally an actor may request a service by placing a pattern in the `requiredbox`. When two actors in an AmbientTalk environment encounter each other, they exchange the patterns from the provided and required services. If a matching service and request is found, the actor requiring the service is notified. The notification transpires by moving the pattern of the relevant service from the `requiredbox` to the `joinedbox`. In ambient environments, network failures or drops between actors is common. The actor requiring the service can be notified if the provider leaves the network by placing the service pattern from the `joinedbox` to its `disjoinedbox`. This alerts the actor that the service is no longer or temporarily unavailable. The actor can then choose to wait till the service

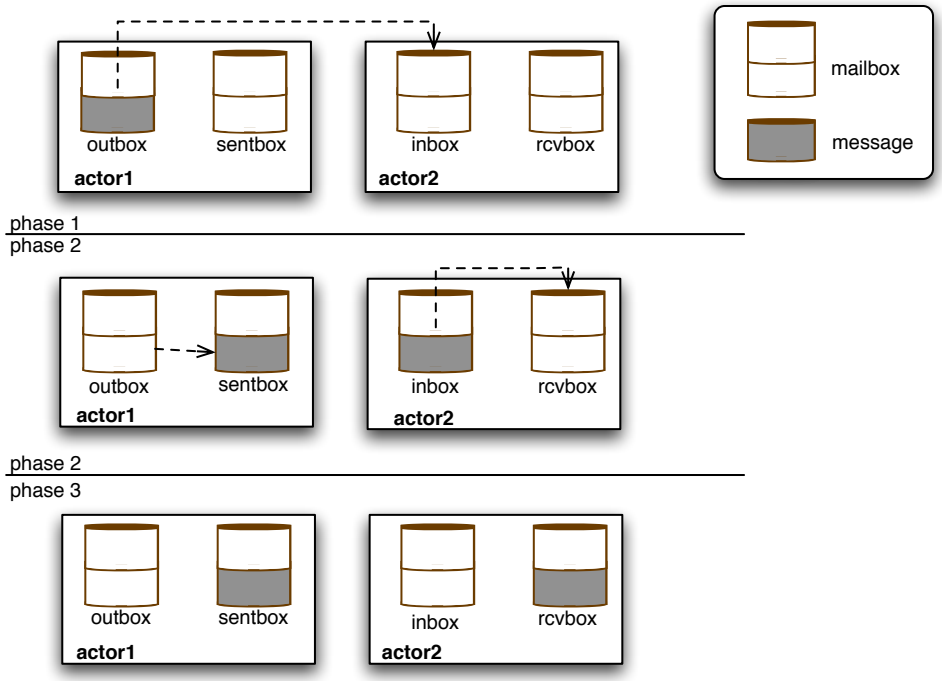

Figure 2.4: A message is sent from the `outbox` of to sending actor to the `inbox` of the receiving actor. Upon successful delivery, the sender moves the message in the `sentbox` mailbox. Meanwhile the receiver may start processing the message. When the message is evaluated, the receiver moves the message from `inbox` to `rcvbox`.

rejoins the ambient environment or request the service anew.

Now that we have introduced AmbientTalk, we can move to a fundamental principle of ambient-oriented programming, named extreme encapsulation.

2.2 Extreme encapsulation

The principle of extreme encapsulation was first introduced by De Meuter in [4] and later explored in [6]. De Meuter argues that language constructs, fundamental to object-oriented languages such as extending objects, do not involve the object in the process. The object has no control over the operation which may disable the object's ability to protect its private information. This may lead to security problems. As a resolution for the security issues inherent from the language, he proposes the extreme encapsulation principle,

> *An object can designate some of its internal state and operations to be private and enforce this property. To be able to uphold this principle, language operators which manipulate an object or class without its explicit intervention are to be prohibited.*

As a solution, De Meuter promotes a system of *method attributes*, based on the Agora model [5]. Method attributes allow objects to give special evaluation semantics to their methods, replacing typical language constructs by special methods in objects. Language constructs are then reduced to message passing to objects. Using method attributes De Meuter states his principle as follows,

> *The principle of extreme encapsulation states that objects should be subject to message passing and message passing alone.*

As motivation, he identifies three language constructs which often breach extreme encapsulation,

1. inheritance, or the extension of objects,
2. the cloning of objects and
3. reflection on objects.

We now elaborate on these examples of language constructs, and show how they violate extreme encapsulation.

2.2.1 Conflict by extension

Prototype based languages such as Self [28] and even AmbientTalk (cfr. section 2.1.1) have an inherent encapsulation problem with extension. This stems from the ability to dynamically extend objects and that the extended objects obtain access to the private state of parent objects. We show this with the following example code[2],

```
LoginSession:object({
  password: "code"
  ...
});
PasswordHacker: extend(LoginSession, {
 showPassword()::{display(password)}
});
PasswordHacker.showPassword(); // --> displays "code"
```

A diagram corresponding to the code is shown in figure 2.5. We have an object `LoginSession` representing the login session for a secure login program. The program creates a login session for each user. The session stores or has access to important user information, such as the password, represented by the variable `password`. In most prototype based languages, one can dynamically extend such an object. A hacker extending `LoginSession` with `PasswordHacker` can supply a method for displaying the password of the parent object. Because `PasswordHacker` has access to the private state of its parent object, it has access to the user's password, and can expose this.

[2]This is AmbientTalk code. It clearly shows that the encapsulation breach by extension has not been resolved in AmbientTalk yet.

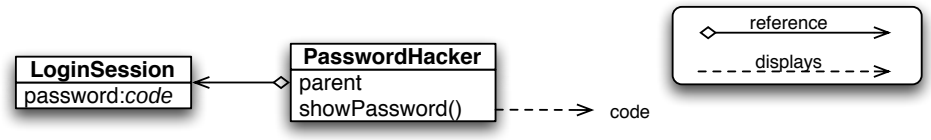

Figure 2.5: In prototype-based languages, objects can be dynamically extended and child objects gain access to the private state of their parent. This is a breach of encapsulation.

It is clear that extension is a possible breach of encapsulation for prototype-based languages. Class-based languages with support for visibility attributes on its members, such as `private` in Java [14], allow a class to explicitly restrict access to those members. In this case, only instances of the class have access to their own copy of that member. Even extended classes do not gain access to fields or methods declared as *private* in its superclass.

2.2.2 Conflict by cloning

Cloning is one of the most fundamental operations in prototype-based languages [24], yet it poses a possible encapsulation breach. As an example, we use again the `LoginSession` object of section 2.2.1 in the following code,

```
LoginSession:object({
  ... as before ...
});
SessionHacker(userSession):object({
 session: clone(userSession);
 accessSystemWithSession()
});
aSessionHacker:SessionHacker(LoginSession);
```

If an external cloning operator is defined, an external client can duplicate an object unhindered. An object `SessionHacker` can then duplicate a user's session (`LoginSession`). While this does not grant `SessionHacker` direct access to private information on `LoginSession`, he has now access to any system using the identity of the user from `LoginSession`.

Prototype-based languages, such as Self [28], resolve this by integrating a default cloning operation in the object. An object can then disable cloning on itself by overriding the clone method. However, a default clone operator does not allow customization of initial values of an object's private state. This means public operations must be provided to set the internal, private, state of an object, breaching encapsulation. We show this with an example (using AmbientTalk syntax) of a bank account in the code below,

```
BankAccount:object({
```

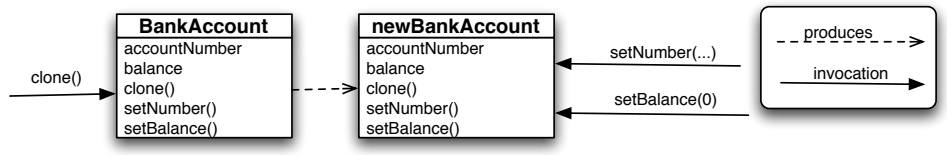

Figure 2.6: Supporting only a default cloning operation in a prototype object forces the inclusion of setters for the internal state of an object so that it may be initialized properly. This can be a breach of encapsulation of an object's private state.

```
 owner: "Alice";
 accountNumber: 100-12345689-24;
 balance: 1.000.000;
 clone()::{... makeCloneOfThis ...};
 setAccountNumber(number):{accountNumber:=number};
 setBalance(amount):{balance:=amount};
 setOwner(aName):{owner:=aName}};

newBankAccount:BankAccount.clone();
newBankAccount.setAccountNumber(101-12345689-24);
newBankAccount:setBalance(0);
newBankAccount:setOwner("Bob");
```

This is depicted in figure 2.6. We see that cloning not only needs explicit interaction with the cloned object, but that a stronger mechanism is required to safely initialize the internal state of the new clone.

AmbientTalk resolves this breach of encapsulation using *cloning methods* as proposed by De Meuter in [4] for ChitChat. Cloning methods are methods with the *method attribute* `cloning` and a special evaluation semantics. When a cloning method is invoked, a clone of the object is created. Additionally, the body of the cloning method is evaluated within the context of the clone. In the body, the private state of the clone can be set, depending on parameters or desired default values. We revise the previous example to include cloning methods,

```
BankAccount:object({
 owner: "Alice";
 accountNumber: 100-12345689-24;
 balance: 1.000.000;
 seeBalance()::{balance}};
 cloning.newAccountFor(aPerson)::{
   accountNumber := aNumber+1;
   balance := 0;
   owner := aPerson;
```

```
 }
newBankAccount:BankAccount.newAccountFor("Bob");
newBankAccount.seeBalance() // --> returns 0
```

Note that we were able to set the internal state of the new object in the body of the cloning method. We do not want the user or creator of the account to simply choose the number or set the balance. Upon creation, however, the name of the owner of the account is required and we can pass that as an argument to the cloning method. This safely imports any external values we require for creating a clone without exposing the private state of the object.

Class based languages such as Java and C++ solve this conflict by having special constructors, and in the case of C++ also with explicit copy constructors. This allows the programmer to explicitly denote how objects can be created and duplicated.

We conclude that an external cloning operator or a fixed cloning operator in an object can breach extreme encapsulation. We have shown that cloning methods allow us to create clones of objects safely without exposing their private state. Cloning methods therefor preserve extreme encapsulation.

2.2.3 Conflict by reflection

The final language operation identified by De Meuter in [4] to typically breach encapsulation is reflection. The breach originates from the reflective operations permitting to see more of an object at its metalevel than at its baselevel. Furthermore the object is seldom involved in controlling the meta-facilities offered by the language. For example in Self [28] the reflective API is based on a technique called *mirrors* [2]. As we explain in more detail in section 2.3, reflection in Self is achieved by external operators which do not necessarily involve the object directly. In fact, their design support the implementation of a different reflection system which may yield more information on objects than originally permitted. This is clearly in violation with the extreme encapsulation principle.

2.2.4 Conclusion

We have shown that important language constructs in prototype-based languages, such as object extension, object cloning and reflection are possible threats for the extreme encapsulation principle. De Meuter [4] claims the breaches stem forth from the lack of an object's involvement in those language constructs. We have seen with cloning methods how methods with special evaluation semantics, give control over the language operation to the object. Method attributes are a convenient way to hand out special evaluation semantics to a method. Using method attributes, language operations such as cloning, can be solved without breaching extreme encapsulation.

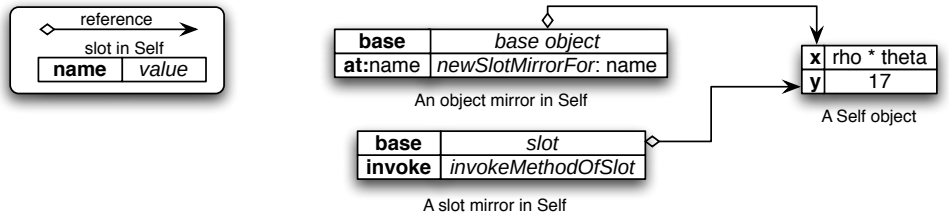

Figure 2.7: In Self, reflection on an object is done through mirrors. Such a mirror can offer access to individual slots by returning a mirror to that slot.

2.3 Mirrors

We now discuss the concept of *mirrors*, a design technique for reflective API's in object oriented languages, introduced by Bracha and Ungar in [2]. The design proposed by Tanter in [27] is based on mirrors as proposed by Bracha and Ungar in [2], which in turn is the starting point for the design of the metaobject protocol proposed in this thesis. Because of the mirror's significance to our design, we discuss the concept of mirrors in more detail and show how they comply with the design principles of Bracha and Ungar. Bracha and Ungar identify three fundamental design principles which any reflective API should obey: encapsulation, stratification and ontological correspondence. Furthermore they propose a design technique for reflective API's, named *mirrors*. Mirrors are a special kind of objects that offer meta-level functionality over base-level objects. In a sense, they encapsulate meta-level facilities over a language element, separating meta-level functionality from base-level functionality. Finally, they argue that mirror-based designs support the three fundamental design principles they identified. We discuss encapsulation, stratification and otological correspondence in further detail in the subsections 2.3.2, 2.3.3 and 2.3.4 respectively, but first we examine the concept mirrors and a mirror-based reflective API.

2.3.1 A mirror-based reflective API

Mirrors were introduced primarily in Self [28] as a reflection mechanism for objects and their slots. The implementation of Self did not support references to slots or methods. The paramount principle of the language is communication between objects, solely by message sends. To support method referencing and invocation, they created objects referring to slots containing methods. These objects could then invoke the method they refer to. This led to the concept of *mirrors*. Mirrors are objects that allow to reason on objects at their meta-level. In Self, a mirror on an object provides access to the slots of that object. Rather than a direct reference to the slot of the object, a mirror referring to the slot is returned. This *slot mirror* then offers an interface for introspection and intercession on its slot (cfr. figure 2.7). From the example of Self, Bracha and Ungar elaborated on the design of mirrors in [2].

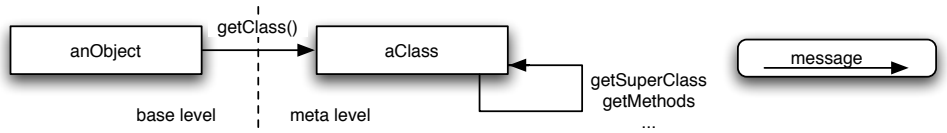

Figure 2.8: In class-based object-oriented languages, the class represents the metalevel of an object and is usually accessible through a message send to the object.

In class-based object-oriented languages, the metalevel of an object is usually accessed via a message to its class such as in Java [12] and Smalltalk [14]. The class of an object is usually accessible through the object using a message send, as depicted in figure 2.8. Reflective operations on classes and their instances are defined at the (meta)class level and coexist with base-level operations side-by-side. Bracha and Ungar deplore this approach and propose the design of mirrors to separate meta-level functionality from the base-level functionality.

We example how mirrors work in a class-based language with the following pseudo-code, taken from [2],

```
class Object{
// no reflective methods
...
}
class Class{
// no reflective methods
...
}
interface Mirror{
String name();
...
}
class Reflection{
public static ObjectMirror reflect(Object o){...}
}
interface ObjectMirror extends Mirror{
ClassMirror getClass();
...
}
interface ClassMirror extends Mirror{
ClassMirror getSuperClass();
...
}
```

We have an object, `anObject` of type `Object`, on which we reflect. First, we obtain a mirror

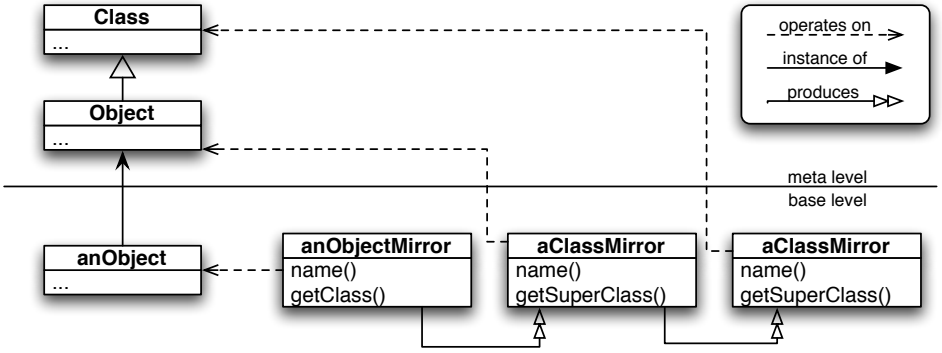

Figure 2.9: Mirrors offer metafacilities on an object, its class and superclass.

to this object using `Reflection.reflect(anObject)`. The object's mirror offers access to the object's class, `Object`. However, it does not return the class itself but a mirror on the class, an instance of `ClassMirror`. The instance of `ClassMirror` in turn offers reflective functionality on the class of the object, such as getting its superclass. This is depicted in figure 2.9.

As we can see, we can access and operate on the metalevel of objects through mirrors. Mirrors control the access to this metalevel, but note that this depends on the implementation of the mirror and that the mirror is defined externally (independently) from the base object. They can offer access to the metalevel of an object's elements, but only through a new mirror. This ensures all reflective operations are done with messages to mirrors. We now elaborate on the design principles, which Bracha and Ungar identified as essential.

2.3.2 Encapsulation

Bracha and Ungar claim that metalevel facilities must *encapsulate* their implementation. This makes it possible to rely on different implementations of the reflective API transparently. Mirrors separate the interface of meta-operations from the implementation of objects. This reduces the interdependence of objects and their reflective facilities. Also, mirrors capture metalevel operations in a separate subsystem. Mirrors propose an interface for reflection inside a language. This allows different reflective systems to be implemented and used transparently. The actual implementation of the mirrors decides how to mirror objects of a given kind, rather than having a fixed reflective system embedded in the language. This is the essence of the first principle in [2], stating that the separation of mirrors, at the meta-level, and classes or objects, at the base-level, is necessary to support the encapsulation of the reflective API.

2.3.3 Stratification

When reflective systems are embedded in the infrastructure of objects and classes, it is difficult to filter out this functionality for applications that do not require reflection. For example, the operations for compiling methods in `Class` in SmallTalk-80 [12] can not be removed since some objects require this functionality and all objects depend on `Class`. The mirror design proposed in [2] separates reflective functions clearly from ordinary functionality. It allows reflection to be loaded or unloaded as an independent subsystem and can thus be deployed depending on the application's requirements. This mitigates an application's footprint, a vital element in an ambient environment, where small platforms are the norm [15].

2.3.4 Ontological correspondence

The principle of ontological correspondence of a reflective API can be split in two parts, structural correspondence and temporal correspondence. To satisfy structural correspondence, the reflective API must reflect on all structural elements of the language. In an object-oriented language, this includes objects and may go as deep as methods and method bodies. Temporal correspondence states that the API must distinguish between static and dynamic properties of the underlying language. Some elements of the language only exist dynamically, such as a message send to an object or the invocation of a function. A complete reflective API should reify such elements and allow metalevel operations on them. As a design suggestion, Bracha and Ungar state that a separable API should be available for reflection on the structural and dynamic or behavioral elements of the language. Using mirrors, the reflective API can be partitioned to offer separate interfaces for structural and behavioral reflection in accordance with structural and temporal correspondence respectively [2].

2.4 Mirror methods

The design of the mirrors proposed in [2] by Gilad Bracha and David Ungar is not compatible with the extreme encapsulation principle [6]. In their design, mirrors on an object are acquired outside the object, using a mirror factory,

```
MirrorFactory.getMirror(obj)
```

With this mirror factory, objects are not directly involved in the creation of the mirror, i.e. they have no control over what *kind* of mirror and *if* a mirror is created. This is in violation with the extreme encapsulation principle, discussed in section 2.2.

In [27], Eric Tanter proposes an alternative design for mirrors in ChitChat [4], in accordance with extreme encapsulation by instating the object with mirror creation responsibility. To achieve this goal, Tanter adapts the technique of *cloning methods*. In

section 2.2.2 we have shown how the method attribute `cloning` denotes a *cloning method*, a method which can create a clone of the receiver in a controlled way. Similarly, Tanter introduces a new method attribute `mirror`, which denotes a new kind of methods: *mirror methods*. A mirror method can a mirror of the receiver in a controlled way. The precise set of metalevel facilities provided by a mirror is then controlled by the receiver itself, since the mirror method determines the expressive power of the mirror, i.e. what it has access to.

The proposal of Tanter in [27] involves two new language concepts, *mirror methods* and a *meta pseudo-variable*. We discuss each element in more detail.

2.4.1 The meta pseudo-variable

Tanter introduces a new pseudo-variable *meta* that refers to the meta-representation of an object. The metaobject exposes an interface to operate on the metalevel of the object: the actual metaobject protocol. The virtual machine is responsible for returning the metaobject, and inherently the meta protocol. Because of its involvement, the reflective facilities depend on the virtual machine. This offers a level of control, a strict virtual machine may limit the protocol, while a more open virtual machine can offer access to other concerns such as garbage collection. Tanter also enstates the following rule,

> *Only mirrors have access to the meta of their creator.*

This rule becomes an important element of our design, as we see in chapter 3.

2.4.2 The mirror methods

To give objects the ability to create a mirror, Tanter proposes a new kind of methods named *mirror methods*. Similar to cloning methods which are denoted by the method attribute `cloning`, mirror methods are denoted by the attribute `mirror`. Mirror methods too, have a unique evaluation semantics, just like cloning methods. When a mirror method is invoked, it first creates a mirror of the receiver and then evaluates the body of the method *in the context of the mirror*. The body of the mirror may declare fields and methods in the mirror, defining its functionality and interface. Mirror methods are in effect constructors for mirrors, just as cloning methods are constructors for clones. The mirror has access to the metalevel of its creator and exposes this using the interface defined by its creator. Since an object defines the mirror constructor and only mirrors have access to the metalevel of the object, mirror methods give objects the control over the meta functionality they expose.

We demonstrate this with an example. Figure 2.10 shows a mirror method `invoker` on points. When invoked, the mirror method `invoker` creates a mirror for `Point` and evaluates its body in the context of the mirror. This declares the only method `methods` in the mirror. The method `meta.methods()` refers to the method table of the receiver. Hence, if `p` is a point, `p.invoker().methods()` returns the table of all the methods of

```
makePoint(aX,aY)::{
   x:aX; y:aY;
   moveTo(nX,nY)::{ x:=nX; y:=nY; }
   mirror.invoker()::{
      methods()::{ meta.methods(); }}
}
```

Figure 2.10: Prototype point object featuring a mirror method.

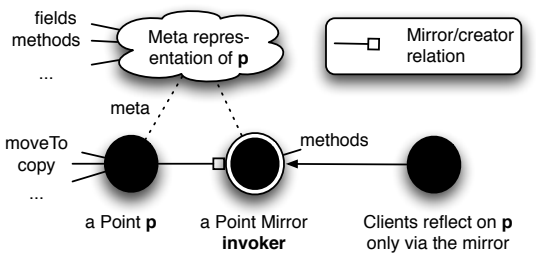

Figure 2.11: Mirror on a point object.

p. One can then use this table to introspect and invoke methods on p reflectively (cfr. figure 2.11).

2.4.3 A structural reflective API

Along with the proposition for the design of mirror methods, Tanter proposed a reflective API in [27] which covers both structural (access to methods and fields) and behavioral (listeners on execution events in the object) aspects.

The structural API offers an interface to reflect on the structure of objects. The function `meta.methods()` from section 2.4.2 is an example of a structural reflective function. However, such a function may not return a direct reference to the methods of the object. Indeed this results in a breach of extreme encapsulation; once a reference to a structural element is outside the mirror, the mirror can no longer control the access to it. Consequently, mirrors must ensure that they return controllable references to structural elements. Tanter solves [27] this, again with mirrors. Rather than a direct reference to, i.e. a field of an object, a mirror to that field is returned, controlling the access to the field. On the other hand, it is undesirable to write a mirror method for each structural element in an object. To avoid writing mirror methods for objects to the extent of fields and methods, Tanter proposes that the structural API never returns a reference to a structural element, but a mirror of that element. For example, the method `meta.methods()` returns a table of mirrors, one mirror for each method. However, different mirrors are possible for a method, i.e. such a mirror may or may not allow introspection of the method's body. To give objects control on the kind of mirror returned, Tanter proposes a number of values which describe the level of access a field or method mirror offers to its structure element. The structural API is then a collection of mirror constructors. We list the suggested functionality,

- `meta.method(name, rights)`: returns a mirror to the method with name `name` and with the access rights described by `rights`.
- `meta.methods(rights)`: returns a table of mirrors, one for each method, and all with the access rights `rights`.

- `meta.field(name, rights)`: returns a mirror to a field with name `name` and access rights `rights`.
- `meta.fields(rights)`: returns a table of mirrors, one for each field, with access rights `rights`

The variable `rights` denotes the access rights on its element the mirror grants. Tanter proposes the following values,

- `meta.read`: the mirror can read the value and name of the field.
- `meta.write`: the mirror may write the value of the field. This right includes the rights of `meta.read`.
- `meta.signatureRead`: the mirror may read the signature of the method.
- `meta.signatureWrite`: the mirror may read and write the signature of the method.
- `meta.invoke`: the mirror may invoke the method reflectively.
- `meta.bodyRead`: the mirror can inspect the body of the function.
- `meta.bodyWrite`: the mirror can inspect the body as well as insert code before or after any instruction.
- `meta.bodyOverwrite`: the mirror can inspect the body, insert any code before or after any instruction and replace any instruction.

We demonstrate the use of rights with the following code,

```
// a Point defintion
makePoint(aX,aY)::{
   x:aX; y:aY;
   moveTo(nX,nY)::{ x:=nX; y:=nY; }
   mirror.inspector()::{
    fieldX()::{meta.field("x",meta.read)}}}
// create a Point
aPoint:makePoint(1,2);
// create a Mirror on the Point
aMirror: aPoint.inspector();
// request a Mirror on the Point's field named "x"
aMirrorOnX: aMirror.fieldX();
// operate on "x"
aMirrorOnX.readValue() // --> returns 1, the value of "x"
aMirrorOnX.setValue(3) // --> error! this is not possible
```

The method `meta.field("x",meta.read)` returns a mirror on the field `x` of the point. Because the mirror was given the permission to read the value but not set it, the mirror will not set the value of the field `x`.

2.4.4 A behavioral reflective API

The behavioral API proposed by Tanter serves to introspect the behavior of an object. Reflection on the behavior of objects is done through the form of a listeners. A mirror can attach (and detach) a listener on language events like field accesses and method invocations. Additionally, they can attach a function to the listener which can be evaluated before or after the event, depending the kind of listener the mirror used. We explain the following code example,

```
createPoint(aX, aY)::{
  ... as before ...
  mirror.moveListener(action())::{
    do()::{action();}
    on()::{meta.before([moveTo], do());}
    off()::{meta.unbefore([moveTo]);}
    on()
 }}
```

We have inserted a mirror method in the point objects, which registers itself on method invocations of `moveTo`. Before the method is invoked, the mirror calls the function `do`, which is bound a given parameter. This allows us to create a tracer, like given in the following code,

```
createTracer(point)::{
  print()::{println("before move");
  point.moveListener(print());
}

p: createPoint(2,3); t: createTracer(p);
p.moveTo(1,1); // this prints "before move"
t.off();
p.moveTo(2,3); // listener is detached, hence no trace
t.on();
p.moveTo(4,5); // listener is attached, this prints "before move"
```

Mirrors can thus activate and deactivate from listening and trigger a callback function on language events. He proposes the following API,

- `meta.before(methods,callback)`: registers the mirror to activate the callback function `callback` before invocations of the methods given in `methods`.
- `meta.before(fields, callback)`; registers the mirror to call `callback` before accesses to the fields given in `fields`.
- `meta.after(methods, callback)`: registers the mirror to call `callback` after message sends to the methods in `methods`.

- `meta.after(fields, callback)`: registers the mirror to call `callback` after accesses to the fields in `fields`.
- `meta.unbefore(methods)`: unsubscribes the mirror to events from methods in `methods`, to which it was registered with `meta.before(methods,callback)`.
- `meta.unbefore(fields)`: unsubscribes the mirror to events from fields in `fields`, to which is was registered with `meta.before(fields,callback)`.
- `meta.unafter(methods)`: unsubscribes the mirror to events from methods in `methods`, to which it was registered with `meta.after(methods, callback)`.
- `meta.unafter(fields)`: unsubscribes the mirror to events from fields in `fields`, to which it was registered with `meta.after(fields, callback)`.

Tanter further proposes to include *context exposure* to enable mirrors to return information of the event to the callback function. Like the structural API, no direct reference may be returned to the callback. Instead, a mirror to the elements on the context is returned and again, a parameter describing the access rights is used. Each element can either be read, or written, thus the rights are limited to `meta.read` and `meta.write`. For the context exposure he proposes the following API for method calls,

- `meta.call(right)`: returns a mirror of a reification of the call.
- `meta.access(right)`: returns a mirror to a reification of the field access.
- `meta.call().receiver(rights)`: returns a mirror to the receiver of the call.
- `meta.call().sender(rights)`: returns a mirror to the sender of the call.
- `meta.call().method(rights)`: returns a mirror to the method.
- `meta.call().args(rights)`: returns a mirror to the argument table.
- `meta.call().result(rights)`: returns a mirror to the result. This is only available after the call.

We demonstrate this API with the following code,

```
createPoint(aX, aY)::{
  ... as before ...
  mirror.moveListener(action(receiver, method))::{
    do(r,m)::{action();}
    on()::{meta.before([moveTo],
      do(meta.call().receiver(meta.read),
            meta.call().method(meta.signatureRead)));}
    off()::{meta.unbefore([moveTo]);}
```

```
    on()
 }}

createTracer(point)::{
  print(receiver, method)::{println("before calling", method.name(),
    " on ", receiver);};
  point.moveListener(print(receiver, method));
}

p: createPoint(2,3); t: createTracer(p);
p.moveTo(1,1); // this prints "before calling moveTo on 2@3"
t.off();
p.moveTo(2,3); // listener is detached, hence no trace
t.on();
p.moveTo(4,5); // listener is attached
     // this prints "before calling moveTo on 1@1"
```

2.4.5 Concluding remarks

The designed proposed by Tanter in [27] reconciles extreme encapsulation for a reflective API based on mirrors. By using mirrors, we have acquired a reflective design with,

- *Encapsulation*: the mirror methods encapsulate the reflective functionality an object uses[3]. This is not entirely in accordance with the encapsulation principle proposed by Bracha and Ungar in [2] which promote complete separation between an application and reflection. But the complete separation is in direct violation with extreme encapsulation.

- *Stratification*: The mirror methods allow to dynamically generate mirrors when needed. An application using mirror methods will therefor only deploy reflective functionality when used. An application without mirror methods will never deploy mirrors. This is in accordance with the stratification principle but also strengthens the encapsulation from above.

- *Ontological correspondence*: The mirror methods proposed by Tanter allow reflection on every language element, such as objects, fields, methods and even method bodies. Also behavioral language constructs such as a method invocation can be reified using mirrors in ChitChat.

- *Extreme encapsulation*: Above all, the principle of extreme encapsulation is respected in the design proposed by Tanter.

[3]With encapsulation we refer to the encapsulation of the reflective API. This is in no way related to extreme encapsulation.

2.5 Summary

In this chapter we reviewed previous research on which we can base the design for a metaobject protocol presented in this thesis. In section 2.1 we have learned that AmbientTalk is a language designed for ambient-oriented programming, which discriminates between two kinds of objects, passive and active objects. Passive objects are regular prototype based objects while active objects follow the ambient actor model presented in [8]. The principle of extreme encapsulation states that objects should be subject *only* to message passing. In section 2.2 we have shown that this principle is often violated by typical language constructs such as extension, cloning or reflection. We have explained how AmbientTalk solves this conflict for cloning using *cloning methods* in section 2.2.2. Mirrors are a design technique for reflective API's introduced by Bracha and Ungar in [2]. We elaborated on how reflective API's based on mirrors have desirable properties for reflections such as encapsulation, stratification and ontological correspondence in section 2.3. In section 2.4 we explain the design approach to mirrors by Tanter in [27]. Tanter shows that the design of mirrors proposed by Bracha and Ungar in [2] violates the extreme encapsulation principle and solves this conflict by introducing *mirror methods*. While mirror methods do not separate the implementation of reflection from an object, Tanter shows that his approach preserves extreme encapsulation and still gains benefits from the mirrors such as stratification and ontological correspondence.
With the information from the previous research presented in this chapter, we can conclude that the design of mirror methods for reflection is a good approach to provide a metaobject protocol that preserves extreme encapsulation. Based on this conclusion we will transmute the design of mirror methods proposed by Tanter to AmbientTalk in the following chapter.

Part II

Passive Mirrors

Chapter 3

Design Of Passive Mirrors

In chapter 2 we have learned that mirror-based reflection using mirror methods preserves extreme encapsulation and still inherits benefits of the mirror approach to reflection such as stratification and ontological correspondence. We have also seen that AmbientTalk distinguishes between passive and active objects. Consequently we split the metaobject protocol presented in this dissertation into a passive and active layer. In this chapter, we focus on the passive layer. We translate the design for a reflective API, proposed by Tanter in [27], to the passive object layer in AmbientTalk.

We start with the design of mirror methods for passive objects, which we call *passive mirror methods* in section 3.1. Subsequently we explain the design of mirrors in the passive object layer, named *passive mirrors* in section 3.2. Finally we give a detailed discussion about reflection on both structural and behavioral aspects of AmbientTalk in sections 3.3 and 3.4 respectively.

3.1 Passive mirror methods

Passive objects in AmbientTalk provide support for cloning operations using *cloning methods*. Cloning methods are methods with a *cloning attribute* and special evaluation semantics. Similarly, passive objects can provide mirroring facilities using *mirror methods*. Much like cloning methods, mirror methods are functions that require special treatment when declared and evaluated. In fact, we may regard mirror methods as constructors of passive mirrors. Upon invocation, the method body of the mirror method is executed in the context of the passive mirror such that variable and constant declarations in the body are stored in the mirror. In this sense, they differ from the evaluation of cloning methods, since cloning methods do not store constants and variables from the method body in the clone. Furthermore, formal parameters of the passive mirror methods become variables in the passive mirrors and are initialized to their corresponding call value.

Passive mirror methods are denoted by their *method attribute*, namely `mirror`. Other

than the method attribute, a passive mirror method resembles a normal method as shown below,

```
mirror.methodname(...arguments...)::{...body...}
```

This declares a mirror method named `methodname` with arguments "`...arguments...`" and body "`...body...`". To explain how we design passive mirror methods, we are first required to describe how its method attribute gives the mirror method its unique evaluation semantics in the AmbientTalk interpreter. However, we will not go into all the details of this evaluation here. We elaborate on that subject in section 4.8. For now we explain the functionality of method attributes at a higher level. A typical method declaration has the following form,

```
methodname(...arguments...)::{...body...}
```

A method attribute, `attribute`, is added before the method declaration as shown below,

```
attribute.methodname(...arguments...)::{...body...}
```

In AmbientTalk, such a declaration undergoes a transformation. The complete transformation is explained in section 4.8. For now, it suffices to know the result of the transformation is the following,

```
methodname :: attribute(methodname(...arguments...)::{...body...})
```

The name of the method is stored, and its value is the result of applying the function named `attribute` on the method. Similarly, the declaration of a passive mirror method with the following form,

```
mirror.methodname(...arguments...)::{ ...body...}
```

is transformed to,

```
methodname :: mirror(methodname(... arguments...)::{...body...})
```

Thus the interpreter binds the name of the mirror method to the result of applying the operator `mirror` on the method. This requires us to implement a special function called `mirror` which transforms a method into a passive mirror method. We elaborate further on the operator `mirror` in section 3.1.2.

3.1.1 Design of passive mirror methods

The evaluation semantics of passive mirror methods differ from normal or cloning methods. We are required to extend the current function model in AmbientTalk to include passive mirror methods. The functionality of a mirror method does not entirely stray from a regular method. In fact, we can break down the invocation process of a mirror method into the following steps,

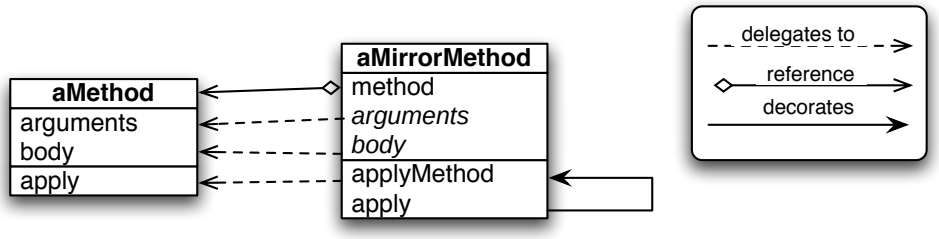

Figure 3.1: A mirror methods is a decorator of a method in AmbientTalk.

1. create a mirror for the current object,
2. bind the actual parameters to the formal parameters *in the mirror*,
3. evaluate the method body *in the mirror*,
4. return the mirror.

Steps 2 and 3 are exactly the same as a normal method invocation, except that the method is executed in the environment of the mirror. Furthermore, the content of the mirror method, that is the list of formal parameters and its body, are the same as that of a normal method. We can consider the evaluation of a mirror method as the evaluation of a normal method, with some addition steps. This lends to a decorator pattern [11], where a mirror method wraps around a normal method and delegates its functionality to the method it contains, adding steps 1 and 4 to the invocation process of the method. This relationship is shown in figure 3.1, where `aMirrorMethod` wraps `aMethod` and `apply` refers to the invocation of the method.

3.1.2 Mirror as native function

The function `mirror` is called for every mirror method declaration (cfr. 4.8 and 3.1.1). Its purpose is to transform a method into a passive mirror method. Since all objects must have access to the function `mirror`, we implement it as a native function, and place it in the native dictionary of AmbientTalk.

3.2 Passive mirrors

The design of passive mirrors is based on mirrors [2], as proposed by Eric Tanter in [27] and discussed in section 2.4. From the elaboration in section 2.4 we conclude that passive mirrors are *normal* objects, except that they *share* the same metalevel as their creator on a conceptual level. A conceptual schema of passive mirrors in AmbientTalk is presented in figure 3.2.

The functionality of mirrors contains two essential elements, the metaobject and the mirror itself. We now explain each element in detail.

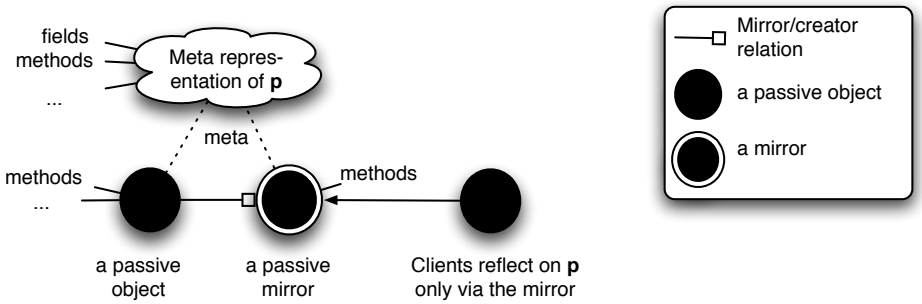

Figure 3.2: A conceptual schema of passive mirrors.

3.2.1 The metaobject

Within the spirit of communicating with objects through message passing, operations on the base object are not done by operating *on* its metaobject but *through* its metaobject using regular message invocation. This approach shares the original vision on mirrors in [2]. The interface offered by the metaobject then forms the actual *metaobject protocol*. We can see the metaobject as a normal object offering (meta-)functions to reflect on the base object. The metaobject holds all metafunctions which operate on an object at the metalevel. While the metafunctions of the metaobject are special, we can regard the metaobject itself as a normal object. However, the construction of the metaobject involves the declaration of many meta functions. We conveniently structure its creation using a *factory pattern* [11]. Furthermore, we offer the same *complete* metaobject protocol to all mirrors. We leave it to the mirrors (and the base objects creating the mirrors with mirror methods) to choose which elements of the metaobject protocol are exposed.

The methods in the *metaobject* operate on the *base object* it represents at the metalevel. This base object must be visible to the metafunctions when invoked. To make the base object visible to these functions, we can use simple scoping rules by declaring a variable *base* in the metaobject. Any mirror has access to a metaobject, which holds the actual metaobject protocol. But in which sense do metaobjects of different objects vary? They offer exactly the same interface; only the base object they operate on differs. Without sacrificing our scoping solution, we want to share this common interface. To include the reference to the base object in our metaobject, we simply extend the metaobject with a new frame and define the reference in the extended metaobject. This leads to the design in figure 3.3 which shows us how metafunctions are shared between different metaobjects.

With this design we create a new metaobject for each mirror using a simple extension.

3.2.2 Meta as pseudo-variable

Mirror methods and the mirrors they create offer a mechanism for reflection which preserves extreme encapsulation by exposing the metalevel of objects in a controlled way (cfr.

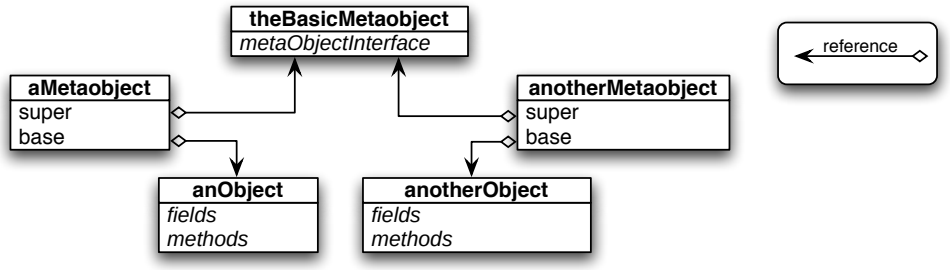

Figure 3.3: The relationship between metaobjects and their shared metafunctions.

section 2.4). Now, we want to void meta exposure, or uncontrolled access to the metalevel of objects. In other words, we want to enforce mirrors and mirror methods as reflection mechanism. To accomplish this, we forbid any other access to the metalevel but through the mirror (and mirror methods). To offer access to the metaobject, we create a fixed reference `meta` in a mirror. Section 2.4.1 argues that *only* mirrors have access to the meta of their creator; that is, not even the creator of the mirror has access to `meta` except through a mirror. We accomplish this control over `meta` in two steps.
First, we do not define `meta` in every object. Only mirrors need access to meta, thus only mirrors need a reference to `meta`. They can, for example, acquire this reference at creation time.
Second, we turn `meta` into a pseudo-variable and allow only message sends to meta. This second step requires modifications in the AmbientTalk parser to treat `meta` as a keyword.

3.2.3 Design of passive mirrors

Passive mirrors are normal objects, except that they share the metalevel of their creator (cfr. section 2.4). The meta level of their creator is represented by the metaobject with a reference to their creator. Creating a mirror is then creating an object with a reference to the metaobject of its creator. The concept of sharing the metalevel insinuates the creator and its mirrors to have a common reference to the same metaobject. However, to enforce the use of mirrors as a reflection mechanism, we allow *only* mirrors to use the `meta` variable (cfr. sections 2.4.1, 3.2.2). We accomplish this, by giving only mirrors a reference to the metalevel of their creator. We can then concretize the conceptual schema of figure 3.2 to the UML-diagram in figure 3.4.

3.2.4 Mirrors on mirrors

At a conceptual level, we consider mirrors as *sharing* their creators metalevel. If we follow our original concept all the way, a mirror created by another mirror, will share the meta of the original base object. It is another mirror of the original object. However, we want the ability to introspect a mirror itself (e.g.: to check which facilities it offers). We rephrase our conceptual design to,

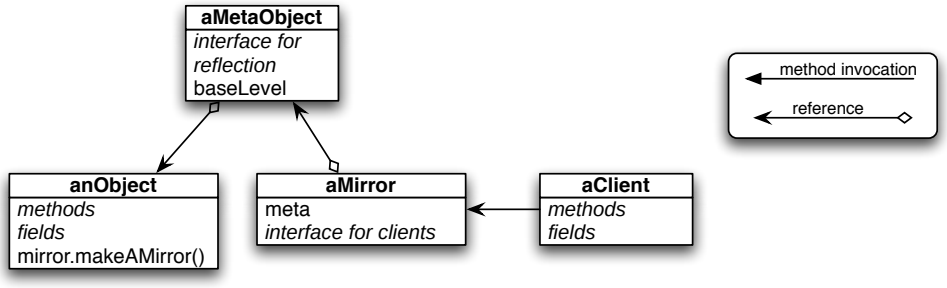

Figure 3.4: A UML representation of the relation between objects, metaobjects and mirrors.

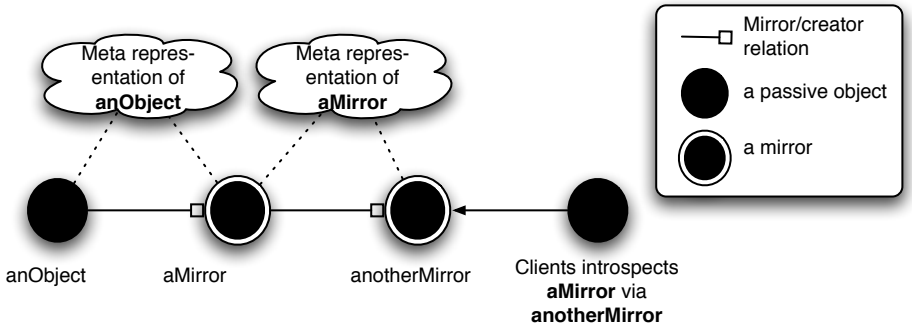

Figure 3.5: A mirror on a mirror.

Mirrors have access to the metalevel of their creator.

This definition allows a mirror to reflect on another mirror. A conceptual schema is depicted in figure 3.5.

But a mirror then has two metareferences, from itself and from its creator. The design presented in 3.2.3 resolves this by allowing a mirror only access to the meta of its creator and not the meta of itself. This means a mirror has *only one* metaobject, the metaobject of its creator. The UML diagram in figure 3.6 concretizes the conceptual drawing in figure 3.5 and shows how our design supports mirrors on mirrors.

3.3 Structural reflection

We have seen in section 2.4 that we reflect on fields and methods. We elaborate on each element separately.

3.3.1 Field mirrors

In section 2.4.3 we introduced an API to reflect on fields. Like reflection for objects, we reflect on a field using a mirror, a *field mirror*. In section 2.1.1 we explained that in

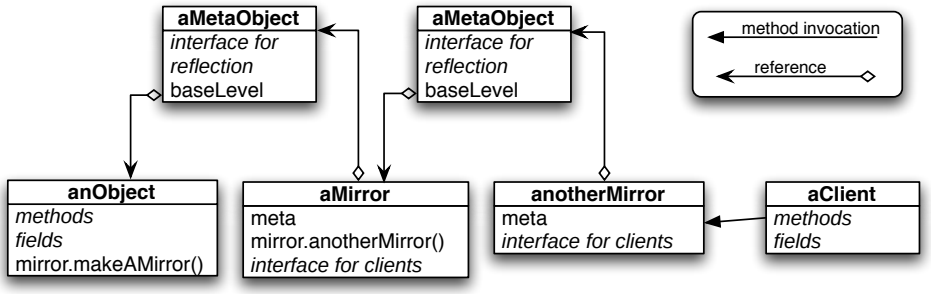

Figure 3.6: UML diagram of a mirror on a mirror.

AmbientTalk, an object is a repository of bindings. Bindings represent the fields on which we wish to reflect. It contains the following attributes,

- `name`: the name of the field.
- `value`: the value of the field.

The metaobject of a field, holds thus a reference to the binding which it represents at the metalevel. Also, the metaobject of a field offers the following operations on the field,

- Read the name of the field.
- Read the value of the field.
- Set the value of field.

The mirror, on the other hand, does not necessarily expose all this operations. We want an object to control the access to its bindings. On the other hand, writing a mirror method for each field we wish to expose is undesirable. In section 2.4.3 we discuss the proposition to include functions such as `meta.field(name,rights)` which generate a field mirror for the binding with name `name`. To control the access rights this mirror offers to clients, the second parameter `rights` describes the access rights the mirror has. We distinguish two policies,

1. a *read* policy offers read access to the name and value of the field,
2. a *write* policy offers write access to the value of the field as well as read access to both the name and the value of the field.

3.3.2 Method mirrors

To reflect on methods, we have introduced a similar mechanism as field mirrors to create mirrors on methods. We call a mirror on a method a *method mirror*. Rather than mirroring the binding of the method, the mirror operates on the method implementation

itself. At the metalevel, a method has a name, a table of arguments and a method body. The metaobject of a method should offer read and write operations for these attributes. Again, the functionality exposed by mirrors should be controlled by the object owning the methods. We provide multiple policies to control the read and write access,

- a *readParameters* policy offers read access to the formal parameters of the method,
- a *writeParameters* policy offers read and write access to the formal parameters of the method,
- a *readBody* policy offers read access to the body of the method,
- a *writeBody* policy offers read access and allows to add statements to the body of the method,
- a *overwriteBody* policy offers read access, allows insertions of statements to the body and allows to remove statements from the body.

The policies of parameters and the method body are independent. To hand-out access to both parameters and body, the object must explicitly note a policy for both parameters and body.

3.3.3 Permissions for mirrors

We have discussed a system of permissions in section 2.4 to describe what a field or method mirror can do. Permissions control the metafunctionality a mirror exposes. This can be accomplished in two ways,

1. **Dynamically**: The field or method mirror stores the given permissions. On each invocation of its methods it checks if its permission allow the invoked method. In case of a violation, it can return either a void value or throw an error. The advantage of this approach is that it is easy to support dynamic changes to the permissions of the field or method mirror. The downside is that at every call a check must be made and the permissions must be stored. Note that to be allowed to change the permissions dynamically, the field or method mirror must be returned to the creator since only the creator can decide which permissions are given.
2. **Statically**: The field or method mirror gets an interface in correspondence to its permissions. For example, if the field mirror does not have the right to *write* the field, the corresponding method for writing fields is not included in the mirror. The advantage of this approach is that no dynamic permission check is needed. If the permission was not granted, the method to perform the operation is not included. To request new permissions, the client must ask a new mirror with the new permissions.

Permissions can only be handed out by the creator, in accordance with extreme encapsulation and the concept of the mirrors as proposed in this thesis. Dynamic permissions can thus only change inside the mirror's creator. This devaluates the approach of dynamic permissions. Hence we opt for the cleaner design technique of the static permissions.

3.4 Behavioral reflection

To reflect on the behavior of objects, we are mainly concerned with accesses to fields and invocations of methods. Field accesses and method invocation are computational events in our language, and using a form of listeners, we can intercept these actions and reflect on them. In section 2.4.4 we introduce the notion of callback functions, the actions performed on such events. Before we dive into design details, we examine the nature of callback functions in more detail.

3.4.1 Mirrors as listeners

Callback functions can be deployed on events such as a field access using the following semantics,

```
meta.before(name, callback)
```

This deploys a listener on the field `name` which triggers the callback function `callback` before the access to the field is executed. Additionally, we may give reified elements of the event to the callback function as arguments. This gives us a first list of requirements,

1. The function representing the callback must be evaluated within the scope of the mirror that deployed the callback.

2. The arguments reifying elements of the event in the interpreter must be evaluated dynamically in the context of the event. The access to these elements is predefined in the mirror deploying the callback function.

3. We must be able to attach listeners to events such as a field access or a method invocation.

4. We must be able to retract any listener previously deployed.

The evaluation of the callback function in the context of its mirror can be solved using a closure. The closure captures the environment of the mirror that deployed the callback function and as such the closure invokes a correct evaluation of the callback function. Arguments passed to the closure are evaluated in the context where the closure is applied. In this context we must offer metafacilities that can return a reification of the event. This requires us to modify the interpreter's evaluation semantics of field accesses and method invocations. The interpreter is required to halt the normal evaluation, check for listeners to the event, place any listeners in the context of the evaluation and trigger their attached closure. A naive approach would be to attach the closures holding the callbacks. If the interpreter offers a context with the required metafacilities for the closure's arguments, this would result in a successful evaluation of the callback. However this discards our ability to retract the callback functions! Once wrapped in a closure, we need to keep a reference

to the closure to ensure a safe detachment of the callback function. This brings additional requirements as to where the references to callbacks must be stored. We propose to reuse the technique of mirrors.

The function `meta.before(name,callback)` creates and returns a mirror holding the callback function. This mirror can be deployed as a listener to events on the field (or method) denoted by `name`. We call such a mirror a *listening mirror*. When an event on `name` occurs, the interpreter places a reification of the event on the stack and invokes the callback functions of any listening mirrors. Since `meta.before(name,callback)` returns a listening mirror for the callback, we have our desired reference on the callback to ensure a safe detachment. The listening mirror offers an interface to dynamically deploy and retract its callback function. We explain listeners on field accesses and method invocations in more detail.

3.4.2 Behavioral reflection on field accesses

A mirror listening to field accesses holds a metaobject offering facilities to the mirror to attach and detach itself from a field. The metaobject is in fact a reification of the metalevel of the field the mirror listens to. In a sense, a mirror listening to field events is a special field mirror. Unlike its structural counterpart, the behavioral field mirror offers an interface to deploy and retract a predefined callback function.

At the metalevel of a field access event, we want a reification of all elements involved in the field access, namely the sender, the receiver and the field being accessed. These can be made accessible by the mirror deploying the callback using parameters for the callback as described in section 2.4.4. Since the callback functions reifies these elements and exposes them to the callback function, the mirror prohibits changing the callback function dynamically, in accordance with the encapsulation principle.

3.4.3 Behavioral reflection on method invocations

For behavioral reflection on methods we apply a similar technique as for behavioral reflection on fields. The function `meta.before(methodName, callback)` deploys a listening mirror with the given callback on events of the method `methodName`. The callback functions can be triggered before, or after the invocation of the method. Parameters reifying elements of the method invocation can be passed as arguments to the callback function as discussed in section 2.4.4.

3.5 Summary

In this chapter we have presented the design for a metaobject protocol in the passive layer of AmbientTalk, based on our findings from previous research in chapter 2. To properly structure the reflective API without breaching extreme encapsulation we used mirror methods to create mirrors (cfr. section 2.4). Mirrors can offer reflective functionality on their creator in a controlled way. In section 3.1 we showed that passive mirror methods are new kind of methods in AmbientTalk which require a special evaluation semantics: upon invocation a mirror method creates a new mirror for the receiver of the invocation and evaluates the method body in the context of the mirror. We also explained in that section that passive mirror methods are created by applying the native functions `mirror` to a method (the mirror method without `mirror` attribute). Then we moved on to section 3.2 where we described that the design of passive mirrors comprises two parts: the design of the metaobject and the design of the mirrors. We chose to model the metaobject as a normal passive object which contains native meta functions. We also designed mirrors as regular passive objects with a unique reference to the metalevel of their creator. In sections 3.3 and 3.4 we elaborated on the metaobjects corresponding to each language construct we reified, such as fields, methods, field accesses and method invocations. We concluded that each metaobject offers a unique set of metafunctions. In the same sections we explained how we need a different mirror for each of the reified language constructs. We explained how they can be seen as normal objects, whose interface is be determined by a set of values describing the access rights the mirror offers on the language construct. Now we can continue to the implementation of passive mirrors and passive mirror methods in AmbientTalk, but first we examine the AmbientTalk interpreter in the following section.

Chapter 4

The AmbientTalk Interpreter

Before we implement the design for passive mirrors proposed in chapter 3, we examine the implementation and internal workings of the AmbientTalk [9] interpreter which is currently written in Java [26]. We begin by identifying the main classes and class hierarchies in section 4.1. Then we explain the most fundamental hierarchies in more detail, such as the `AbstractGrammar` hierarchy in section 4.2, the value hierarchy of `AGValue` in section 4.3 and the hierarchy of functions in AmbientTalk in section 4.4. Subsequently we explain some other fundamental classes in the interpreter: continuation frames, represented by `ContinuationFrame`, in section 4.5 and native functions in section 4.6. With this information we explain briefly the principle idea of the *read-eval-print*-loop in section 4.7. We conclude this chapter with an example; we evaluate a *cloning method* and observe how it is processed inside the interpreter in section 4.8.

4.1 The main hierarchies

The internal mechanics of the AmbientTalk interpreter are characterized by the following classes or class hierarchies,

- **Parser:** is the parser for AmbientTalk expressions. It can parse a Java *String* representing an AmbientTalk expression and returns an abstract syntax tree where *AbstractGrammar* is the common superclass of all the tree's elements.
- **AbstractGrammar:** is the abstract superclass of all grammar objects in the interpreter. All instances of this class are first-class citizens in a running AmbientTalk application.
- **ContinuationFrame:** is the abstract superclass of all continuation frames in the interpreter. They embody the continuation stack. Each continuation frame defines an order of evaluation on its elements, which are instances of *AbstractGrammar*.
- **Process:** is an abstract superclass for all processes in the interpreter such as the main process of the local virtual machine and the process controlling the actor threads. It

keeps a reference to all information describing a process's state like the current active environment, continuation stack etc.

- **ProcessEvalThread:** represents a Java thread and defines the evaluation cycle of an AmbientTalk expression. It also offers some rigid control over the evaluation process such as killing the process with the *killEvaluation* method.
- **PicoCallBack:** is responsible for returning information to the user, which can be text, grammar objects or notifications.

When working at the level of the interpreter, it is important to keep the interactions of these different classes in mind. Some of the class hierarchies states above will require modification when we implement our metaobject protocol. We examine the class hierarchies, most relevant to our design, in more detail below.

4.2 The AbstractGrammar hierarchy

The *AbstractGrammar* class is the common superclass for all first-class objects that make up an AmbientTalk application. While every subclass embodies another element of the application, *AbstractGrammar* shows us some common functionality used throughout the program,

- the evaluation interface, *eval()*, which requests the grammar object to evaluate itself, possibly pushing new *ContinuationFrames* on the stack,
- basic type checking, i.e.: *isDictionary()*,
- basic type conversion, i.e.: *asDictionary()* which is also used to cast an object's class in a controlled way.

It is important to note that every language construct we will reify in our reflective API will be a subclass of `AbstractGrammar`. In chapter 6, where we introduce reflection on the active layer of AmbientTalk, we will encounter the one exception to this rule, namely an actor's thread. We discuss that is more detail in chatpers 6 and 7.

4.3 The AGValue hierarchy

One subclass of *AbstractGrammar* we are particulary interested in is *AGValue*. It is the superclass for all grammar objects we work with in the AmbientTalk language,

- passive objects (*AGDictionary*),
- active objects (*AGActor*),
- primitive types such as numbers (*AGNumber*), texts (*AGText*) and void (*AGVoid*),

- functions (*AGFunction*), which we discuss in more detail in 4.4,
- native functions (*AGNative*).

Subclasses of `AGValue` are values which can be returned in AmbientTalk. It is evident that our mirrors, metaobjects and even the metafunctions will be subclasses of `AGValue`, but not necessarily direct subclasses as will become clear in chapter 5.

4.4 Function implementation

In the AmbientTalk interpreter, functions are represented by *AGFunction* grammar objects, a subclass of *AGValue*. Each subclass of *AGFunction* represents a different kind of function and function evaluation, [1]

1. **AGFunctionImpl:** represents normal functions of AmbientTalk,
2. **AGCloningFun:** represents cloning methods.

Instances of *AGFunction* function in two steps. Firstly, the function object is created with all its *static* information (formal parameters, body, name). This object is the result of a function definition and can be passed around in AmbientTalk (ie. can be assigned to a reference). Secondly, the object can be activated with all the *dynamic* information (actual parameters, dynamic environment) using the *apply(...)* method. This corresponds to an invocation of the function in AmbientTalk.

4.5 Continuation frames

Continuation frames represent frames on the continuation stack. Each frame has a number of grammar objects and the frame defines an order of evaluation on these grammar objects. The common superclass *ContinuationFrame* provides the following,

- the methods *cntContinue, cntPoke, cntProceed, cntPush, cntReturn* for the basic continuation stack behavior,
- a value holding an *AbstractGrammar* object which is evaluated at creation time of the *ContinuationFrame*, and can be passed to the next continuation frame the result of the *ContinuationFrame*,
- the *execute()* interface, which triggers the continuation frame to execute.

[1] In fact, we extend *AGFunction* once for passive mirror methods, once for active mirror methods as explained in sections 5.2 and **??** respectively.

4.6 Native functions and thunks

Native functions in AmbientTalk are written in the native language of the interpreter (Java [26]) and require a special evaluation. A single grammar object, namely *AGNative*, is used to wrap any native function and represent it in AmbientTalk. The implementation of native functions involves three classes. We explain their role,

- **AGNative:** Wraps and represents a native function in AmbientTalk. It can be called like a function, with all dynamic information (actual parameters, dynamic environment) for the evaluation. It passes this information to the *Native* object which it wraps.
- **Native:** is the superclass of native functions inside the interpreter. It does not belong to the grammar hierarchy. The *Native* class provides common functions for counting and evaluating the arguments. Subclasses of *Native* can directly evaluate their function (i.e. if no arguments need to be evaluated) or request the evaluation of their arguments and send the resulting parameters to a *Thunk*.
- **NativeMetaThunk:** is the superclass of all thunks. Thunks are used for all functions that expect their arguments to be fully evaluated. These thunks are called when all arguments are evaluated and used to avoid having a specialised continuation frame for all natives, that now use thunks.

4.7 The Read-Eval-Print loop

The read-eval-print loop is defined by *ProcessEvalThread*, *Parser*, *AbstractGrammar*, *ContinuationFrame* and *PicoCallBack* following these steps,

1. *ProcessEvalThread* receives the request to evaluate a *String* and passes this to *Parser*.
2. *Parser* parses the *String* and returns an *AbstractGrammar* object representing the AmbientTalk expression.
3. *ProcessEvalThread* initializes the continuation stack with the bottom continuation frame *CFEnd* and places the AbstractGrammar object on the continuation stack.
4. *ProcessEvalThread* requests the *AbstractGrammar* object to evaluate itself.
5. The grammar object evaluates, thereby (possibly) pushing new continuation frames on the stack to evaluate subexpressions.
6. *ProcessEvalThread* requests the top frame of the continuation stack to execute its evaluation (in a loop).
7. When *ProcessEvalThread* receives a signal that the evaluation is complete, it outputs the obtained result to *PicoCallBack*.

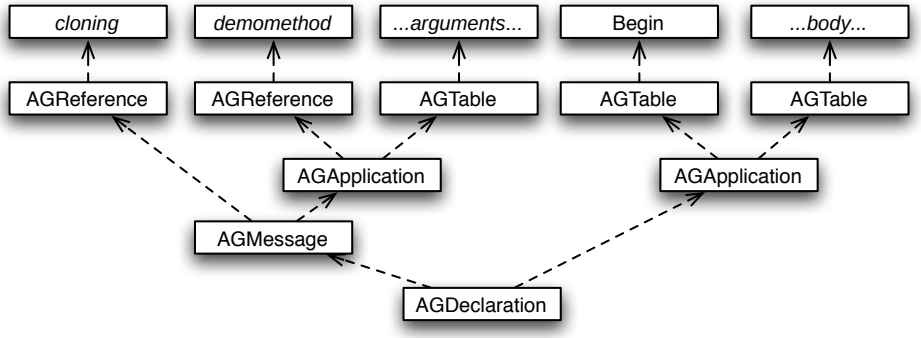

Figure 4.1: The parse-tree returned from parsing a cloning method.

4.8 Example: declaring and evaluating a cloning method

To show how the interpreter works, we observe an example. As example we take the declaration of a *cloning method*. We feed the interpreter the following code,

```
cloning.demomethod( ... arguments ... )::{
 ... body ...
}
```

Parser parses the *String* which and returns a declaration object (*AGDeclaration*). This represents the parse-tree depicted in figure 4.1. The body of the function is considered an application of the function *begin* on the table (*AGTable*) containing the statements of the body.[2] Note that the declaration binds an application (*AGApplication*) to a message send (*AGMessage*). This is because code of the form *object.message(parameters)* is considered a message send. Why the parser allows this kind of declaration becomes more apparent as we continue the evaluation process.

ProcessEvalThread initializes the continuation stack with the bottom frame (*CFEnd*) and the top node of parse-tree is evaluated,

```
topOfStack = new CFEnd(parseTree);
parseTree.eval();
```

The sequence diagram of the above steps is shown in figure 4.2

A sequence diagram of the following steps is given in figure 4.3. The declaration object (*AGDeclaration*) evaluates by propagating the request to the left hand of the declaration. In our example the message (*AGMessage*) receives that request. If the left-hand expression of the declaration is a normal reference, this would result in a normal declaration but *AGMessage* treats this differently. It puts a special continuation frame (*CFEvalMsgDeclReceiver*) on top of the continuation stack which evaluates the receiver, that is *cloning*. *cloning* is implemented as a native function. Thus we have,

[2] *Begin* is a native function which evaluates a table of expressions.

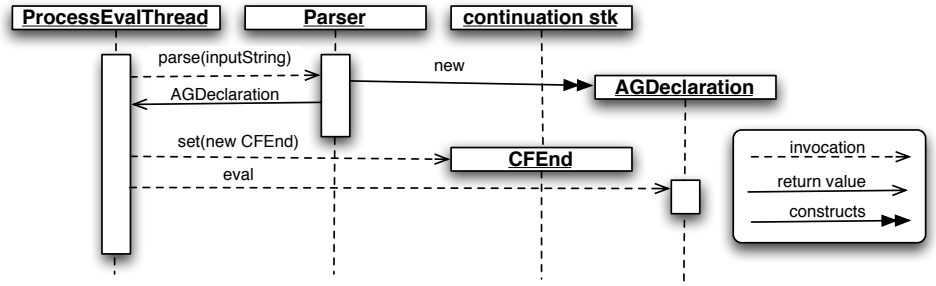

Figure 4.2: The initial steps for evaluating the declaration of a cloning method.

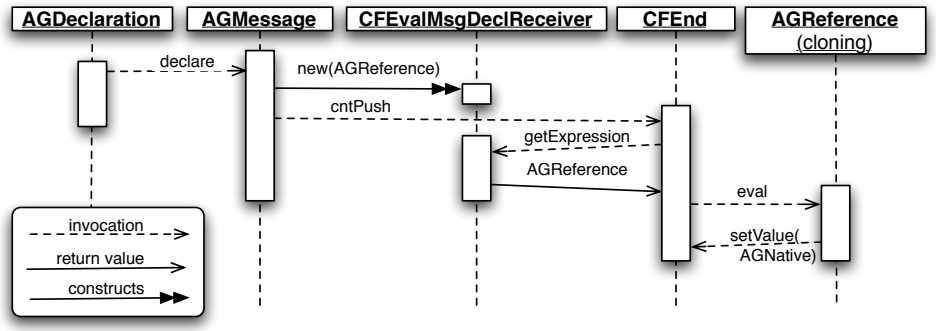

Figure 4.3: The evaluation of the *cloning* attribute in the declaration.

- **NATCloning:** as subclass of *Native*, which implements the native function *cloning*,
- **AGNative:** representing *NATCloning* in the native dictionary.

Searching *cloning* yields the native (*AGNative*) holding the *cloning* function.

Since the evaluation stopped, *ProcessEvalThread* requests the top continuation frame (*CFEvalMsgDeclReceiver*) to execute. The application (*AGApplication*) representing *demomethod(...arguments..)* is required to handle the declaration. However, the application *knows* it has to declare a message of the form,

```
attribute.message(...arguments..)::{...body...}
```

And it will transform this expression in to the following,

```
message :: attribute(message(...arguments...)::{...body...})
```

In short, the function *demomethod* is bound to the result of applying the native function *cloning* to it. Naturally, *cloning* transforms a function into a *cloning function*. This process is shown in the sequence diagram in figure 4.4. The final steps of the evaluation are quite trivial,

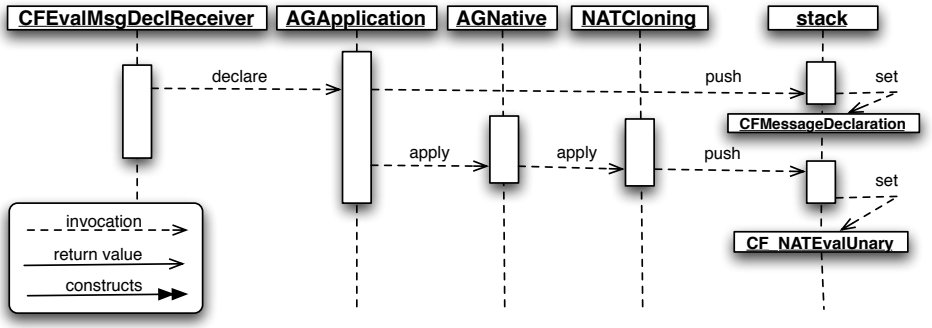

Figure 4.4: The sequence of evaluating the cloning method.

- the application inserts a continuation frame (CFMessageDeclaration) which will bind the result of the *cloning* function to the reference *demomethod*.
- the function *cloning* asks *Native* to evaluate the given function implementation.
- *Native* puts a continuation frame (*CF_NATEvalUnary*) on the stack to evaluate the function.
- the top continuation frame (*CF_NATEvalUnary*) evaluates the function, asks *cloning* to process the function (into a cloning function) and puts the result on the stack.
- the next (top) continuation frame binds the cloning function to *demomethod*.
- the remaining bottom continuation frame (*CFEnd*) signals the end of the evaluation.
- *ProcessEvalThread* returns the result to the user.

4.9 Summary

In this chapter we have seen an overview of the implementation of the AmbientTalk interpreter in section 4.1. We have identified and elaborated on the hierarchy of abstract grammar objects in section 4.2, of values in AmbientTalk in section 4.3 and of functions in 4.4. In the latter we have seen how all functions in AmbientTalk have a common interface `AGFunction` and every *kind* of method is a subclass of `AGFunction`. With a kind of method, we refer to the different kinds of evaluation semantics. At this moment, the AmbientTalk has only two kinds, normal methods and *cloning methods*. In section 4.5 we have shown that continuation frames define an evaluation order on the grammar objects and in section 4.6 we explained the implementation of *native functions*. Native functions are subclasses of a *non-grammar* object `Native`. Such a subclass of `Native` is then wrapped in the abstract grammar object `AGNative` which represents native functions in

the language. Also we have identified *thunks*. Thunks are small functions which accept evaluated arguments an return the value of the function they implement. Many native functions are reduced to thunks. Subsequently, we explained the *read-eval-print*-loop of the AmbientTalk interpreter inb section 4.7 and example how the interpreter functions by observing the interpreter when evaluating a *cloning method* declaration. We have seen how the interpreter transforms an expression of the form,

```
attribute.methodname(...args...)::{...body...}
```

to an expression of the form,

```
methodname::attribute(methodname(...args...)::{...body...})
```

As a consequence, cloning methods are created by applying the native function `cloning` to a method declaration.

Now that we have seen the essential elements of AmbientTalk's interpreter, which play an important role in the evaluation and declaration of methods with attributes, we can progress to the implementation of passive mirrors and passive mirror methods. We elaborate on the implementation in the following chapter, based on the design proposed in chapter 3.

Chapter 5

Implementation Of Passive Mirrors

In chapter 2 we have examined preceding research which have led to a set of design principles on which we based the design proposed in chapter 3. In chapter 4 we have studied the interpreter of AmbientTalk and the evaluation semantics of method attributes. We are ready to implement the proposed design of chapter 3. We begin with the implementation of the method attribute `mirror`, which denotes a *mirror method*, in section 5.1. We explain how the `mirror` attribute enables the creation of *passive mirror methods*. The implementation of passive mirror methods is discussed in section 5.2 and we show how they generate *passive mirrors*. We elaborate on the implementation of a mirrors `meta` variable in section 5.3 and show how we transform `meta` into a pseudo-variable. In section 5.4 we show that the metalevel of different language entities, such as fields and methods, can actually benefit from a similar design. We then apply this common design for the metalevel of each language construct in the subsections of section 5.4. To fully support the reflective API as described in chapter 3, we need a more powerful system to search fields or methods in an object, which we introduce in section 5.5. Finally in section 5.6 we give an overview of the possible passive mirrors that can be created using passive mirror methods and the reflective API implemented in this chapter.

5.1 The native function mirror

From section 3.1.2 and the elaboration in section 4.8 we know that mirror methods are created by applying the native function `mirror` on a normal method. This transforms the given method into a mirror method. Since `mirror` is native, we implement it as a subclass of `Native`, called `NATMirror`. Any native function (or subclass of `Native`) is called using `apply` and the following arguments,

1. **arguments:** a table of arguments (AmbientTalk elements) passed to this function. For the native function `mirror`, it should hold one argument, the method to be transformed.

2. **evalDct:** the dictionary in which to evaluate the function. In the case of a mir-

ror method declaration, this corresponds to the object in which the passive mirror method is declared.

3. **thisDct:** the currently active environment. Again, for `mirror`, this is the object in which the passive mirror method is declared.

Only one argument is required for `mirror`, the method that must be transformed. The method is not yet evaluated when `mirror` is called; `NATMirror` must ensure the method's evaluation. `NATMirror` calls `Native` to evaluate the argument and pass the result to the thunk `THKMirror`, which executes the transformation,

```
public class NATMirror extends Native {
...
public void apply(AGTable arguments, AGDictionary evalDct, AGDictionary thisDct)
     throws PicoException, Signal {
          Native.evalOneArgument(arguments, THKMirror._instance);
}
...
}
```

`THKMirror`, given a method, returns the corresponding passive mirror method using the following code extract,

```
if (method.isFunction()) {
   return new AGMirrorFun(method.asFunction());
} else if (method.isClosure()) {
  return new AGMirrorFun(method.asClosure().getFunction());
} else throw new NativeException(
   "mirror expects a function or a closure as argument");
```

where `AGMirrorFun` represents a mirror method, as we explain in the following section.

5.2 Implementation of passive mirror methods

Passive mirror methods introduce a new kind of function. We subclass the abstract superclass of all functions, `AGFunctions` (cfr. section 4.4), with `AGMirrorFun`. Normal functions in AmbientTalk bind their formal parameters to the corresponding actual parameters and allow declarations of variables and constants. Passive mirror methods differ only in the environment in which the function is evaluated, namely the mirror. This lends to a *decorator pattern* for implementing mirror methods: `AGMirrorFun` accepts a normal function, adds some behavior to the `apply` method before it forwards the call to the function it holds. In this manner, `AGMirrorFun` regulates the environment for the normal function it invokes. The resulting function hierarchy is shown in figure 5.1.

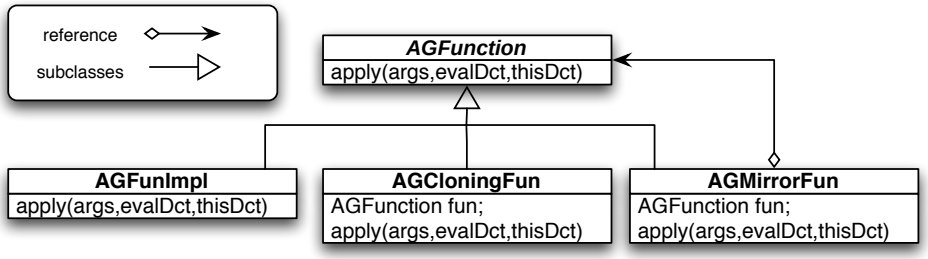

Figure 5.1: Passive mirror methods decorate a normal function in AmbientTalk.

A call in the interpreter to the method `apply` of `AGMirrorFun` corresponds to a method invocation of the passive mirror method in AmbientTalk. It requires the following steps,

1. Creating the passive mirror.
2. Evaluating the function arguments, binding them in the mirror. Any required references not defined in the mirror should be looked up in its creator.
3. Force the mirror as return value of the mirror method instead of the last result of the method.

The code from `AGMirrorFun.apply` for these steps is given below,

```
// create the mirror
AGDictionary mirror = MirrorFactory.newMirrorFor(creator);
// evaluate the function arguments, bind them in the mirror,
// search in the mirror's creator: the receiver of the method.
call(arguments, formalParameters, method, mirror, creator, superDct);
// return value of the mirror is always the mirror itself
topOfStack.insert(new CFForceReturn(mirror,
   currentDictionary,
   thisDictionary,
   superDictionary));
```

Note that in the code we use a mirror factory. This does not correspond to the mirror factory technique proposed by Bracha and Ungar in [2]. Their approach was built in the language, while our technique is at the level of the interpreter and our language constructs are *safe*.

In the AmbientTalk interpreter, the binding of the actual parameters to the formal parameters and the evaluation of the body is the responsibility of `AGTable.call`. This method's first step is to extend the current dictionary with a new frame, the function's environment. However, in mirror methods we have already created our mirror environment

and don't want the environment to be extended[1]. We implement a new version of this method as `AGMirrorFun.call`.

Binding parameters and evaluating the method body is executed by the continuation frame `CFStartBindings`. `AGMirrorFun.call` pushes this continuation on the stack,

```
topOfStack.poke(new CFStartBindings(
  actuals,
  formals,
  mirror, // without extending the environment!
  methodBody,
  creator,
  supDct,
 0));
```

5.3 Implementation of meta as pseudo-variable

To offer access to the metaobject, we create a fixed reference `meta`. Section 2.4.1 argues that *only* mirrors have access to the meta of their creator; that is, not even the creator of the mirror has access to `meta` except through a mirror. We accomplish this control over `meta` in two steps. First, we do not define `meta` in every object. Only mirrors need access to meta, thus only mirrors receive a reference to `meta`. They acquire this reference at creation time, shown in section 3.2. Second, we turn `meta` into a pseudo-variable and allow only message sends to meta, as explained below.

Avoiding exposure of meta

Mirrors enable a safe means to expose and operate on the meta-level of an object. We enforce this mechanism by making `meta` a pseudo-variable, in accordance to proposed design of section 2.4.1. Also, we allow only message sends to `meta`. AmbientTalk offers no support for such restrictions so we extend the language to treat `meta` as a keyword.

We change the scanner of AmbientTalk (`Scanner`) to recognize `meta` as a keyword and return the new token *Token.MET*. We disallow any operation on meta but message sends. In `Parser`, we force this by parsing a message send when a meta-token is received. The code for this operation is given below,

```
private static AbstractGrammar readOperand() throws ReadException {
switch (_currentToken) {
 // added to restrict meta usage to message sends.
 case (Token.MET) : { return readMetaInvocation(); }
```

[1]If we would, after the evaluation of the method, the extended (function's) frame would be removed and all the defined variables and constants with it. This would leave us with an empty mirror.

```
 ... same as before ...
 }
}
 // added for meta invocations
private static AbstractGrammar readMetaInvocation() throws ReadException {
 AGReference meta = new AGReference(TextPool.allocate("meta"));
 skip();
 switch(_currentToken){
  case (Token.PER):{ return readInvocation(readQualification(meta)); }
  default: {
   throw new ParseException("meta can only accept message sends.");}
}}
```

This successfully restricts the use of `meta` to message sends. Also, it avoids that the meta object can be returned as a value. In fact, it cannot be referenced in any way but through message sends inside the mirror. This is exactly the kind of safety we desire.

5.4 The family of metaobjects

In chapter 3 we described a whole family of metaobjects. Before describing each of them, we identify their common structure and consequently their common design.

5.4.1 Parents and their children

While the metafunctions of the metaobjects are special, the metaobjects themselves behave as normal objects and are used with message sends. We implement meta objects as normal passive objects. The construction of the metaobjects involves the declaration of many native functions. To conveniently structure its creation we use a *factory pattern* [11]. In section 3.2.1 we discussed how to use extension as code-sharing mechanism between metaobjects of the same kind. We also identified a kind of metaobject for each of the following entities,

- basic objects,
- fields,
- methods,
- field listeners,
- method listeners.

Each kind of these metaobjects can use the code-sharing technique we described. The different kinds of metaobjects differ only in the interface they contain. Therefor we write a common abstract factory class that singles out the instantiation of a kind-specific factory

class to an initialization method, the method for loading the interface of the metaobject.

We call the abstract superclass `MetaFac`. It provides the following common interface,

```
public abstract class MetaFac {
...
abstract protected void initMeta()
  throws DictionaryException;

public AGLocalDictionary getSharedMeta()
  throws DictionaryException{
...
}
public AGDictionary getNewMeta()
   throws GrammarException, DictionaryException{
  ...
}
public AGLocalDictionary newMetaFor(AbstractGrammar obj)
  throws GrammarException, DictionaryException{
  ...
}
public AbstractGrammar getBaseFrom(AGDictionary meta)
 throws DictionaryException{
  ...
}
}
```

5.4.2 The meta of objects

The functions in the *metaobject* operate on a *base object*. This is the object the metaobject represents at a metalevel. It offers the following interface,

- `field(name,rights)`: returns a field mirror of the field with name `name` and with the permissions `rights`.
- `fields(rights)`: returns a table of field mirrors with the permissions `rights`, one for each field in the base object.
- `fields(namepattern, rights)`: returns a table of field mirrors with the permissions `rights`, one for each field in the base object that matches the pattern `namepattern`.
- `method(name,rights)`: returns a method mirror on the method with name `name` and permissions `rights`.

- `methods(rights)`: returns a table of method mirrors with the permissions `rights`, one for all the methods in the base object.
- `methods(namepattern, rights)`: returns a table of method mirrors with the permissions `rights`, one for each method with a name matching the pattern `namepattern`.
- `before(namepattern,callback)`: returns a table of listening mirrors with callback function `callback`, one for each field with a name matching the pattern `namepattern`. The field mirrors are attached to their field, and listen to accesses on the field.
- `before(tableofnames, callback)`: returns a table of listening mirrors with callback function `callback`, one for each name in the table `tableofnames`. The field mirrors are attached to their field, and listen to accesses on the field.
- `unbefore(tableofmirrors)`: detaches every listening mirror in the table, `tableofmirrors`, that was deployed using `before`.
- `unbefore(mirror)`: detaches the listening mirror, `mirror`, that was deployed using `before`.
- `after(namepattern, callback)`:returns a table of listening mirrors with callback function `callback`, one for each field with a name matching the pattern `namepattern`. The field mirrors are attached to their field, and listen to accesses on the field. The callback is invoked after the field access.
- `after(tableofnames, callback)`:returns a table of listening mirrors with callback function `callback`, one for each name in the table `tableofnames`. The field mirrors are attached to their field, and listen to accesses on the field.
- `unafter(tableofmirrors)`:detaches every listening mirror in the table, `tableofmirrors`, that was deployed using `after`.
- `unafter(mirror)`:detaches the listening mirror, `mirror`, that was deployed using `after`.

The parameters that can be passed to the callback functions are described below.

5.4.3 The meta of field accesses

The meta of field accesses is rather slim. It offers the following interface,

- `access.sender(rights)` returns a field mirror on the reference to the sending or querying object of the field access, with the rights described by `rights`.
- `access.receiver(rights)` returns a field mirror on the reference to the receiving object of the field access, with the rights described by `rights`.
- `access(rights)` returns a mirror with access to both the sender and the receiver of the field access. The permission on the object are set by `rights`

5.4.4 The meta of method invocations

The meta of method invocations offers the following interface,

- `call().sender(rights)` returns a field mirror on the reference to the sender or querying object of the method invocation, with the rights described by `rights`.
- `call().receiver(rights)` returns a field mirror on the reference to the receiving object of the method invocation, with the rights described by `rights`.
- `call().method(rights)` returns a method mirror on the method being invoked, with the rights described by `rights`.
- `call().args(rights)` returns a table of mirrors on the actual arguments of the method invocation, with the rights for all fields described by `rights`.
- `call().result(rights)` returns a field mirror on the result of the method invocation, with the rights described by `rights`.
- `meta.call(right)` returns a mirror offering access to all elements of the invocation with permissions `rights`.

5.4.5 The meta of fields

The metaobject of fields holds all the meta-operations for fields. This includes,

- `readName()` returns the name of the field.
- `readField()` returns the value of the field.
- `writeField(value)` sets the value of the field to `value`

5.4.6 The meta of methods

The metaobject of methods holds the following metafunctions,

- `readName()` returns the name of the method.
- `readParameters()`: returns a copy of the parameter table of the method.
- `writeParameters(parameterTable)`: sets the parameter table of the method to `parameterTable`.
- `readBody()` returns a copy of the table of expressions of the method.
- `writeBody(newBody)` sets the body of the method to `newBody`.

5.4.7 The meta of field listeners

- `before()` attaches the mirror to the field to trigger the callback before evaluating the field access.
- `unbefore()` retracts the mirror from the field if registered with `before`.
- `after()` attaches the mirror the field to trigger the callback after evaluating the field access.
- `unafter()` retracts the mirror from the field if registered with `after`.

5.4.8 The meta of method listeners

- `before()` attaches the mirror to the method to trigger the callback before evaluating the method invocation.
- `unbefore()` retracts the mirror from the method if registered with `before`.
- `after()` attaches the mirror to the method to trigger the callback after evaluating the method invocation.
- `unafter()` retracts the mirror from the method if registered with `after`.

5.5 Filtered searching

The reflective API in section 2.4 describes functions creating mirrors of methods or fields. While fields can hold any value, from function to integer, a method is always a function. Note that a field holding a function is actually a method. To find a method, we must not only check the name of the field but also its value. The search functions in dictionaries provide only support for name-based searching. We create a new class `FilterSearch` which can scan the bindings of dictionaries and store any binding that validates according to a validation function. The validation function is abstracted using a strategy pattern. Since a validation function operates on the binding, or field, of an object, we can create different semantical searches. In our implementation we provided the following,

- fields
- methods
- fields corresponding to a name query
- methods corresponding to a name query

For name queries or name patterns we allow *strings* with the following syntax,

- a letter in the string represents exactly the same letter,

- a `?` represents any letter,
- a `*` represents any amount of any letter,
- a `.` denotes a method attribute. We can write a name pattern on each side of the `.`.

This allows us to search on fields with name queries as demonstrated below,

```
meta.fields("var*")
```

This returns a table of mirrors on all fields with a name that begins with `var`. Another example,

```
meta.methods("mirror.*")
```

This returns a table of mirrors, one for each *mirror method*.

5.6 The family of passive mirrors

From the elaboration in section 3.2 we conclude that passive mirrors are *normal* objects, except that they hold a reference to the metalevel of their creator. The metalevel of their creator is represented by the metaobject with a reference to their creator. Thus creating a mirror is creating an object with a reference to the metaobject of its creator. Some mirrors have a default implementation while the interface of other depends on the rights granted for the mirror. We find that different kinds of mirrors differ only in the metaobject they hold and their interface. Similar to the metaobjects, we create an abstract factory class to facilitate the implementation of mirrors using factories.

5.6.1 Passive mirrors for objects

Passive mirrors are normal object, except that they keep a share the meta level of their creator (cfr. section 2.4). We extend the abstract factory for mirrors to return *empty* mirrors with only a reference to the meta of their creator. The following code excerpt shows the method for creating a new mirror,

```
... in MirrorFactory ...
public static AGDictionary newMirrorFor(AbstractGrammar base)
 throws GrammarException, DictionaryException{
     AGDictionary mirror = newDictionary();
     mirror.addVariable(MetaFactory._textForMeta, MetaFactory.newMetaFor(base));
     return mirror;
}
```

where `MetaFactory._textForMeta` represents the textual reference used for `meta` in the interpreter. We obtain the design shown in figure 3.4. The final step of creating a mirror is evaluating the mirror method's body in the context of the mirror, so that variables and the interface of the mirror can be declared, cfr. section 5.2.

5.6.2 Field mirrors

In section 2.4.3 we introduced an API to reflect on fields. To protect a fields baselevel and metalevel, we want to reflect over the field using mirrors, which we call *field mirrors*. Field mirror allows operations on the field, through the metalevel, in a controlled way. Possible operations a field mirror can offer are,

- `name()`: returns the name of the field.
- `read()`: returns the value of the field.
- `write(value)`: sets the value of the field to `value`.

Field mirrors are created using the field mirror factory.

5.6.3 Method mirrors

To reflect on methods, we have introduced a similar mechanism to create a mirror on a method. We call a mirror on a method a *method mirror*. Rather than mirroring the binding of the method, the mirror operates on the method implementation itself. At creation time, we give the method mirror the following interface,

- `name()`: returns the name of the function.
- `readParameters()`: returns a copy of the list of parameters for this function.
- `writeParameters(listOfParameters)`: sets the method's list of parameters to `listOfParameter`
- `readBody()`: returns a copy of the method's body.
- `writeBody(newBody)`: sets the method's body to the closure `newBody`.

5.6.4 Listening mirrors for fields

We discriminate between two kinds of field listeners, those that apply their callback with `before` and those that apply their callback with `after`. The first kind has the following interface,

- `name()`: returns the name of the field they listen on,
- `before()`: if the callback function is not deployed, it will now deploy the callback to react before access to the field is made,
- `unbefore()`: if the callback function is deployed, it is now retracted from the field.

Field listeners created with the `meta.after` method offer the following interface,

- `name()`: returns the name of the field they listen on,

- `after()`: if the callback function is not deployed, it will now deploy the callback to react after an access to the field is made,
- `unafter()`: if the callback function is deployed, it will be retracted from the field.

5.6.5 Listening mirrors for methods

Similar to listening mirrors for fields, we distinguish between the listeners created with `meta.before` and the ones created with `meta.after`. They offer exactly the same interface as the listening mirrors made with `meta.before` or `meta.after` respectively.

5.7 Summary

We implemented the method attribute `mirror` as a *native function* which transforms a method declaration into a mirror method declaration (cfr. section 5.1). We extended the function hierarchy of AmbientTalk with *mirror methods*. Mirror methods add some extra functionality to a normal method and are therefor implemented as a *decorator* for a normal method (cfr. section 5.2). Mirror methods create passive mirrors, which keep a reference to the metalevel of their creator. This reference is not publicly accessible. We have restricted all access to `meta` by declaring it as a pseudo-variable in section 5.3. In section 5.4 we showed that the metalevel of all language elements we reflect on can be represented by normal objects, although they contain native metafunctions depending on the language element they represent. subsequently we have defined a new means of searching for fields and methods in an object in section 5.5. By using a strategy pattern to qualify matching bindings, we are capable of evaluating matching fields or methods on more than just their name, for example we showed the use of a name pattern. Finally in section 5.6 we elaborated on the interface a mirror can have for each language element we offer reflection on.

The implementation presented in this chapter realizes the proposed design of chapter 3. We evaluate the implementation in chapter 8. This completes the metaobject protocol for the passive layer proposed in this thesis, however this is not necessarily a complete metaobject protocol. AmbientTalk has other language elements which we have not reified in the metalevel yet, and offered no mirrors for. We discuss possible extensions of this metaobject protocol as future work in section 9.2.

Part III

Active Mirrors

Chapter 6

Design Of Active Mirrors

In this chapter we propose our design for reflection in the active object layer of AmbientTalk [9]. We have seen in section 2.1.2 that the active object layer of AmbientTalk is populated by actors. We identified the three main characteristics of an actor to be its behavior, mailboxes and unique thread. In our metaobject protocol we want to reflect on these elements using the design techniques of mirrors, presented by Bracha and Ungar in [2], and mirror methods, presented by Tanter in [27].

We begin by identifying the properties of actors on which we reflect in section 8.2. In section 6.2 we show how we adapt the technique of passive mirror methods from section 3.1 to a similar kind of methods for active objects, called *active mirror methods*. We continue with the design of *mirrors* in the active layer, which we call *active mirrors*, in section 6.3. Finally, we describe a reflective API for both structural and behavioral aspects of active objects in sections 6.4 and 6.5.

6.1 Reflection in the active layer

In section 2.1.2 we explained that the active layer of AmbientTalk is populated by actors that communicate via asynchronous message passing as opposed to synchronous method invocation occurring at the passive object layer. Passive objects expose their meta facilities through passive mirrors. Similarly we want active objects to expose their meta facilities through *active mirrors*. Active mirrors follow the same design principles of mirrors, proposed in [2], but exist in the active object layer. This means they are active objects, or actors. Like passive mirrors for passive mirror methods, active mirrors can only be created by invoking *active mirror methods* on an actor. This allows an actor to selectively hand out different kinds of active mirrors on itself.

We now explore which metafacilities an active mirror can offer by examining the structure of actors. In section 2.1.2 we have identified the following three characteristics of actors,

- actors have a unique passive object which defines its *behavior*
- actors posses four *mailboxes* for communication purposes and four *mailboxes* for the service discovery mechanism
- actors occupy exactly one *thread*.

When designing the reflective API for actors, it is important that we reify those three characteristics at the structural and behavioral level. We discuss reflection on each of these elements in more detail below.

6.1.1 Behavior

The behavior of an actor is defined by a single passive object (cfr. section 2.1). An active object should be able to offer structural information of its behavior object in the metaobject protocol. Because the behavior is a passive object, reflection on the behavior can partially be accomplished in cooperation with passive mirrors. The behavior is, however, relevant to reflection in the active layer and we need a mechanism to make the behavior accessible in active mirrors. By exposing the behavior object in the metalevel of an actor, an actor can offer reflection on its behavior in the mirror.

6.1.2 Mailboxes

An actor contains eight mailboxes, four for communication purposes and four for the service discovery mechanism (cfr. section 2.1.2). By keeping a history of their communication, actors have a reified communication tree. When actors have a joint operation, such a communication tree allows them to recover from, for example, an inconsistent state between them. Inspecting the communication tree means introspecting the mailboxes. This is a crucial application of reflection on the mailboxes. In a similar way, reflection on the service mailboxes allows actors to introspect their own or another's services. Also, introspection of service mailboxes may greatly facilitate the service discovery mechanism. For example, an actor requiring a communication service may place the pattern `"communication"` in his `requiredbox`. When an actor offering `"communication"` is in the ambient environment, the requesting actor is notified. But there is a potential problem with this strategy. The pattern `"communication"` is a rather vague description which can be interpreted in different ways. For example one could offer a communication service over a WiFi or BlueTooth network. How does the actor requiring a communication service decide which service to select? At best, he can request a service for each variant of the communication service and decide which to select when at least one service is connected. It is clear that a unique name for each of these services could make the program overly complex. Introspection of an actor's service mailboxes allows an actor to check and decide dynamically if the target actor offers an interesting service and place a matching pattern in his `requiredbox` to acquire that service.

In the metalevel of actors, we include a reficiation of the meta of its mailboxes. The actor can then offer introspection on, for example, what services an actor provides or requires. Typical operations on the metalevel of mailboxes are,

- introspection of its contents,
- adding or removing messages in communication mailboxes,
- adding or removing patterns in service mailboxes.

We include a reification of the metalevel of mailboxes and the above operations in our metaobject protocol.

6.1.3 Thread

An actor embodies a unique thread (cfr. section 2.1.2). A reification of the thread at the metalevel of an actor would offer control over the execution process of an actor. For example, we could ask the thread to pause, continue or to evaluate a single message. This can serve as basis to support extensive debugging, for example by forcing a step-by-step execution of the thread. With a step we refer to a computational step at the actor level, or the processing of a single message from the `inbox` of the actor. Consequently, operations on an actor's thread at the metalevel should offer,

- pausing the thread,
- continuing the thread,
- evaluating the next message in the inbox of the actor.

We now have identified what elements of an actor we reflect on and which facilities on these elements we desire. We begin the design of our metaobject protocol in the active object layer below.

6.2 Active mirror methods

In section 3.1, we show how passive mirror methods allow the creation of passive mirrors in accordance with extreme encapsulation [4]. Similarly to passive mirrors method for passive objects, we design active mirror methods for active objects. Active mirror methods create active mirrors, an *active* object which has access to the metalevel of its creator. Active mirror methods allow an actor to expose its metalevel in a controlled way. We discuss active mirrors in more detail in section 6.3.3. Now we examine how they can be created using active mirror methods.

6.2.1 Design of active mirror methods

Active mirror methods are methods (of an actor) with the method attribute `amirror`. When such a method is invoked, it creates a new active mirror of the receiver which contains a reference to the metalevel of the receiver. The body of the method defines the behavior of the active mirror. In this manner, actors can selectively expose their metalevel using active mirror methods. This approach resembles passive mirror methods greatly but there are some important differences,

- an active mirror method does not produce a passive, but an active mirror,
- the active mirror is an actor and must be deployed in the ambient environment correctly.

Like passive mirror methods, active mirror methods have unique evaluation semantics. We describe their evaluation procedure as follows,

1. Upon invocation, a new object must be created. This object will become the behavior object of the active mirror.
2. The behavior object obtains a reference to the meta level of the receiver of the invocation.
3. The actual parameters of the active mirror method are bound to the formal parameters of the active mirror method, *in the new behavior object*.
4. The method body of the active mirror method is evaluated *in the context of the behavior object*. This ensures that variable and method declarations in the method body are declared in the mirror's behavior.
5. The behavior object is returned as a result from the mirror method.
6. An actor must be created and deployed with the constructed object as its behavior.

We design active mirror methods as a new kind of functions in the interpreter with these evaluation semantics. Equivalently to passive mirror methods, active mirror methods wrap a normal method and extend only the behavior of the method application. We design them as decorator of a normal method as shown in picture 6.1

6.2.2 The native function amirror

In sections 3.1.1 and 4.8 we have seen how a declaration of the form,

```
amirror.methodname(...args...)::{...body...}
```

is transformed to,

```
methodname:: amirror(methodname(...args...)::{...body...})
```

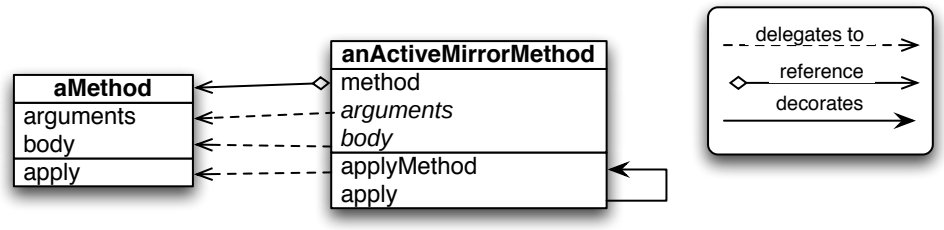

Figure 6.1: An active mirror method wraps a normal method and extends only the behavior of the method application. Active mirror methods follow the decorator pattern for methods.

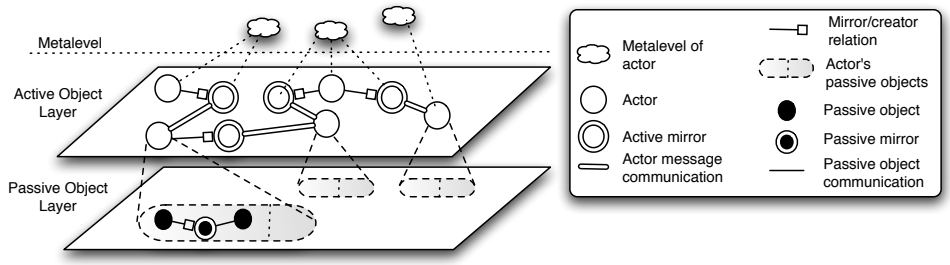

Figure 6.2: The different layers of AmbientTalk objects and their mirrors.

Like `mirror` for passive mirror methods, `amirror` is a function which transforms a normal method into an *active mirror method*. The function `amirror` is available to all actors in the interpreter, so we design `amirror` as a native function and place it in the native dictionary of AmbientTalk.

6.3 Active mirrors

Mirrors operate on the same level as their creators. Passive mirrors are deployed in the passive object layer and likewise we deploy active mirrors in the active object layer of AmbientTalk. This gives us the conceptual representation of the different layers and their corresponding mirrors in figure 6.2

Active mirrors are in effect normal actors in AmbientTalk, except that they contain a reference to the metalevel of their creator, called the *meta actor*. The key elements of our active mirror design are the reference to the meta actor and the active mirrors holding this reference. We discuss each element in detail below.

6.3.1 The metaobject of actors

The metaobejct of an actor offers the interface to operate on the actor at its metalevel. In section 8.2 we have explained the metalevel of an actor includes its behavior, mailboxes and thread. On each of these elements, the metaobject protocol should offer operations. But

where should the operations on the metalevel be executed? The behavior and mailboxes are passive objects in the actor. In correspondence with the encapsulation principle, these may only be accessible in the context of the actor to which they belong [6]. Furthermore, an actor embodies a thread which exists only in the context of the actor. We conclude that the operations on the metalevel of an actor are evaluated *within the context (and thread) of the actor itself*. We explain two possible approaches to achieve this,

1. We create the metaobject of the actor in the context of its mirror and the metaobject keeps a reference to the base actor. This design is similar to the design for metaobjects in the passive object layer and looks attractive at first but there is a problem with this approach. The base actor is a different actor than the active mirror. All operations provided by the metaobject correspond to a metaoperation on the base actor. These functions must communicate *asynchronously* with the base actor, operate on the metalevel in the context of the base actor, and return the result to the mirror. This forces us to implement futures at the level of the interpreter. Furthermore, each function offered by the metaobject must regulate its communication with the base actor. If several operations on the metalevel must be performed sequentially, the active mirror waits between each operation for the previous to complete and return a result. This can cause a time and communication overhead.

2. We create the metaobject in the context of the actor. Functions offered by the metaobject of the actor operate on the metalevel of the local actor. But then, an active mirror must communicate asynchronously with the metaobject of the actor. When a series of operations must be performed, the mirror can send a *block of code* in the form of a function to the metalevel of the base actor. The base actor can then evaluate this function in its meta context. If the mirror desires a result, it can attach a statement to send a message to itself with the desired result.

We have opted for the second approach for its simplicity and to avoid problems like time and communication overhead. We allow mirrors to send codeblocks in the form of functions to the metalevel of the base actors. They are evaluated in a context with access to the metalevel of the actor. The code blocks a mirror can create depends on its behavior, and its behavior is described by the base actor in the mirror method. This preserves extreme encapsulation.

In our design, we make a distinction between the metalevel of the actor *inside the actor* and access to this metalevel *inside the mirror*. This is accomplished with two different metaobjects. The first metaobject represents the metalevel of the actor in the context of the actor. It is a passive object that offers operations on the actor at its metalevel. An active mirror can refer to this metaobject in the code blocks using `meta`. The second metaobject is an actor reference. It refers to the metalevel of an actor inside the context of the actor; that is the context in which the first metaobject is accessible. Mirrors gain a unique reference to this metalevel, named `metaActor`.

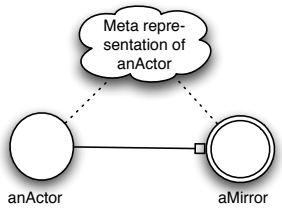

Figure 6.3: The relation between an active mirror and its creator.

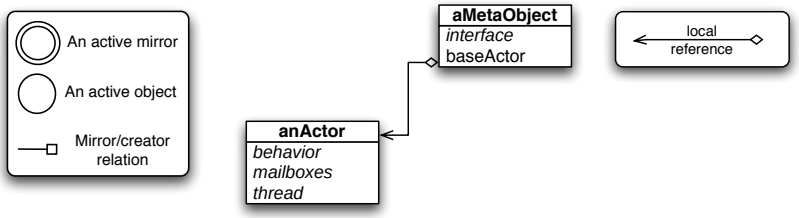

Figure 6.4: The metaobject of an actor is stored locally.

6.3.2 The metaActor pseudo-variable

Active mirrors hold a reference to the meta of their creator, named `metaActor`. Equivalently to passive mirrors, we enforce active mirrors as a reflection mechanism by limiting the use of `metaActor` to (asynchronous) message sends. We accomplish this with the following steps,

1. We transform `metaActor` into a pseudo-variable.
2. We allow only *asynchronous* message sends to `metaActor`.
3. Only active mirrors obtain a reference to `metaActor`.

Note that message sends to `metaActor` are asynchronous. This is because active mirrors are independent active objects, just like their creator.

6.3.3 Design of active mirrors

Active mirrors are, in the first place, active objects. They differ from normal actors in the sense that they are created by active mirror methods and have access to the metalevel of their creator. A conceptual representation of the relation between an active mirror and its creator is shown in figure 6.3.

In section 6.3.1 we explained that the metaobject of an actor is actually local to the actor. We show this in figure 6.4. An active mirror does not hold a reference to the metaobject, but accesses it through the meta actor. The meta actor keeps a reference to the metaobject of the base actor. The UML diagram in figure 6.5 shows this relation.

In this way, the active mirror can execute operations at the metalevel of an actor.

6.4 Structural reflection for actors

We now describe the metalevel facilities we offer for each element of the actor.

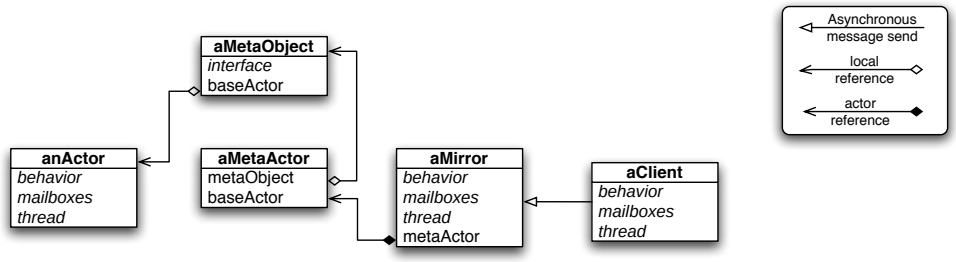

Figure 6.5: The meta actor is an environment wrapped around the base actor.

6.4.1 Behavior mirrors

The behavior of an actor is described by a unique passive object. We can access this behavior object in the metalevel of the actor using the following operation,

```
meta.behavior()
```

Unlike most meta functionalities, we do not return a mirror here but the behavior object itself. Because the behavior is a passive object, it can only be used with its public interface. Further reflection on the object can be offered using passive mirrors and passive mirror methods in the object itself.

6.4.2 Mailbox mirrors

Mailboxes of actors are first-class citizens of the AmbientTalk interpreter. All eight mailboxes are instances of the same class in Java. This means that we can offer the same metaobject to operate on all mailboxes of the actor. Mailboxes can be obtained in the metalevel of the actor using the following command,

```
meta.getMailbox(name,rights)
```

This returns a mirror on the mailbox, `name`, with the rights policy denoted by `rights`. The name can be any of `inbox`, `rcvbox`, `outbox`, `sentbox`, `providedbox`, `requiredbox`, `joinedbox` or `disjoinedbox`.

The rights policy can be any of the following,

- `meta.read` : allows to introspect the contents of the mailbox
- `meta.add` : allows to add messages or patterns to the mailbox
- `meta.delete` : allows to remove messages or patterns from the mailbox.
- `meta.modify` : allows to modify messages or patterns from the mailbox.

Messages can be added using the command,

```
mailboxMirror.addMessage(sender, method, arguments)
```

which adds the message to execute `method` with the arguments denoted by `arguments`. For service mailboxes we can add patterns using,

```
mailboxMirror.addPattern(actor, pattern)
```

6.4.3 Message mirrors

Reflection on a mailbox never returns a message itself. It returns a mirror on the message, which has access to the message on its metalevel. The metalevel of a message offers the following methods,

- `getReceiver()`: returns the receiver of the message,
- `getSender()`: returns the sender of the message,
- `setSender(actor)`: sets the sender of the message,
- `getMessage()`: returns the name of the method the message will execute,
- `setMessage(method)`: sets the name of the command the message will execute,
- `getArguments()`: returns the table of arguments from the message,
- `setArguments(arguments`: sets the table of arguments from the message

6.4.4 Thread mirror

A mirror on the thread of an actor can be obtained using the command,

```
meta.getThread()
```

We not include a policy for thread manipulation. This is because the operations on the thread are limited to the essential operations such as pausing and continuing the actor's thread. The interface of a thread's mirror offers the following,

- `pause()`: pauses the current actor. Only messages to the metalevel of the actor are still processed. This guarantees that mirrors will still be able to tell the actor to continue its normal execution.
- `eval()`: if paused, this operation enables a mirror to eval the next message in the actor's inbox. This is required for step-by-step execution
- `continue()`: This command will tell the actor to continue its normal execution.

6.5 Behavioral reflection for actors

Behavioral reflection on the actor can be accomplished in two ways, by introspecting its behavior object or by listening to actions on its mirrors. Listening to actions on the actor's behavior object is done with the passive mirrors techniques from chapter 3. We now explain how we can listen to actions on the mailboxes.

6.5.1 Behavioral reflection on mailboxes

We distinguish two kinds of actions, the addition of a message or pattern to the mailbox and the removal of a message or pattern from the mailbox. We offer the following interface,

- `onAddition(mailbox, callback)`: returns a listening mirror that will trigger the callback function `callback` when an item is added to the mailbox.
- `onRemoval(mailbox, callback)`: returns a listening mirror that will trigger the callback function `callback` when an item is removed from the mailbox.

The interface of listening mirrors on mailboxes is similar to listening mirrors on fields and methods. They offer the following interface,

- `name()`: returns the name of the mailbox they listen on,
- `start()`: if the listening mirror is not deployed, it will now be attached to listen to actions on the mailbox,
- `stop()`: if the listening mirror is deployed, it will now be retracted from the mailbox.

6.6 Summary

In this chapter we explained reflection on active objects in AmbientTalk. Active objects have three prime characteristics on which we reflect, their behavior, their mailboxes and their thread. At the metalevel of actors, we reify those three characteristics. To properly structure the reflection without breaching extreme encapsulation we proposed active mirror methods to create active mirrors (cfr. section 6.2). Like passive mirror methods, active mirror methods are a decorator of a normal method. In the interpreter, an active mirror method is created using the native function `amiror` which transforms a normal method into an active mirror method. Active mirror methods allow the construction of active mirrors in a controlled way. Active mirrors are actors which have access to the metalevel of their creator. However, unlike passive mirrors, active mirrors can not posses a direct reference to the metalevel of an actor as this would be in violation with extreme encapsulation. As a workaround we store the metalevel of an actor in the actor itself. An active mirror is then given unique access to this metalevel through a reference called the *meta actor*, denoted `metaActor`. The `metaActor` lets an active mirror execute code blocks in the form

of functions at the metalevel of its creator *asynchronously* (cfr. section 6.3). The kind of functions it can execute is entirely dependent on its behavior and thus the creator's mirror method. This solution preserves extreme encapsulation. Other than their unique access to their creator's metalevel, active mirrors are considered normal active objects.

In sections 6.4 and 6.5 we elaborated on the metaobjects corresponding to each element of an actor we made accessible in the metalevel. This includes the actor's behavior, mailboxes and thread. We concluded that each metaobject offers a unique set of meta-functions. Also, for each element of the actor we defined a systematic way to constructs mirrors for the element. This allows us to expose the metalevel of an actor's characteristics in a controlled way. We now continue and explain the implementation of the reflective API we have proposed in this chapter.

Chapter 7

Implementation Of Active Mirrors

In chapter 6 we have presented a design for reflection on actors based on active mirrors and active mirror methods. In this chapter we elaborate on the implementation of the designed. We start with the native function `amirror` in section 7.1, and explain how it actually transforms an implemented method into and implemented active mirror method from section 7.2. We show how we restrict access to the meta actor reference in section 7.3. In section 7.4 we elaborate on the metaobjects for actors, their behavior, their mailboxes and their threads. Finally, in section 7.5, we explain the implementation of the different mirrors, active or passive, we created to safely expose the metaobjects of section 7.4.

7.1 The native function amirror

When an active mirror method is declared, the native function `amirror` is called to transform the declared method into an *active mirror method*. We explain the implementation of the active mirror method in section 7.2. Now we explain how they are created using the operation `amirror`. Since `amirror` is available to all actors, it is stored as a native function in the native dictionary of AmbientTalk. We implement it as a subclass of `Native` called `NATAMiror`. It is invoked at the level of the interpreter with the method `apply` and the following arguments,

1. `arguments`: a table of arguments passed to the function. In the case of `amirror` this contains one argument, the method to be transformed into an active mirror method.

2. `evalDct`: the evaluation dictionary of the arguments. For `amirror` it contains the actor's behavior object.

3. `thisDct`: the current active environment. Again, this is the actor's behavior object for the function `amirror`.

The only argument `amirror` requires is the method that must be transformed into an active mirror method. Firstly, this method must be evaluated, so `NATAMirror` asks `Native` to evaluate the method and send the result to the thunk `THKAMiror`,

```
public class NATAMirror extends Native {
...
public void apply(AGTable arguments, AGDictionary evalDct, AGDictionary thisDct)
   throws PicoException, Signal {
        Native.evalOneArgument(arguments, THKAMirror._instance);
   }
...
}
```

`THKAMirror`, given the method, transforms it into an active mirror method as follows,

```
if (method.isFunction()) {
   return new AGAMirrorFun(method.asFunction());
 } else if (arg.isClosure()) {
   return new AGAMirrorFun(method.asClosure().getFunction());
} else {
  throw new NativeException("active mirror expects
    a function or a closure as argument");
}
```

where `AGAMirrorFun` represents an active mirror method as explained below.

7.2 Implementation of the active mirror methods

Active mirror methods are a new kind of functions in AmbientTalk. They have a unique evaluation semantics, so that they create active mirrors. Consequently, we implement them in a new class named `AGAMirrorFun` which we make a subclass of `AGFunction`, the abstract superclass for all functions in AmbientTalk. In section 6.2.1 we have shown the different steps of the evaluation semantics of an active mirror. Active mirror methods differ from normal methods in the environment in which the method body is evaluated, namely the behavior object of the active mirror. Next, they are responsible for creating an actor with the behavior object created by the method. They can be seen as *decorators* of normal methods as we see in picture 7.1.

Invoking an active mirror method requires the following steps,

1. Creating the behavior object of the active mirror.
2. Evaluating the function arguments, binding them to the formal parameters in the behavior.
3. Evaluating the method body in the behavior, allowing the declaration of methods and variables.
4. Adding the `metaActor` reference.

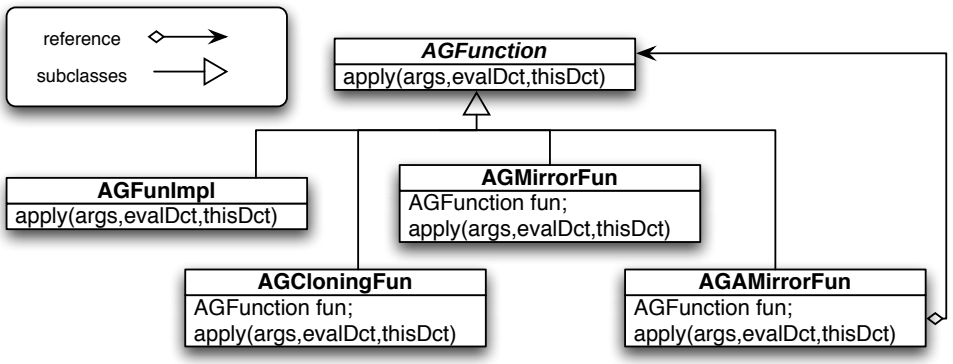

Figure 7.1: Active mirror methods are decorators of a normal AmbientTalk function.

5. Creating an actor with the behavior object, the actual active mirror.
6. Returning the active mirror as result from the function.

This is accomplished with the following code excerpt from `AGAMirrorFun.apply`,

```
// create a new (behavior) object
AGDictionary behavior = new AGLocalDictionary(rootDictionary);
// evaluate the function and its arguments, bind them in the mirror,
// search in the mirror's creator: the receiver of the method.
call(arguments, formalParameters, method, behavior, receiver, supDct);
// add reference to metaActor
behavior.addVariable("metaActor",metaActor);
// initialize the actor
AGActor amirror = new AGActor(behavior, Device.DEVICE, true);
amirror.initializeActor(true);
// return the actor
topOfStack.insert(new CFForceReturn(amirror,
 currentDictionary,
 thisDictionary,
 superDictionary));
```

Similar to passive mirror methods, calling the function in the behavior object does not require the environment to be extended. We use the same `call` function used for passive mirror methods.

7.3 Implementation of metaActor as pseudo-variable

To give an active mirror access to the metalevel of its creator we give it a fixed reference named `metaActor`. Similar to `meta` for passive mirrors, we want to avoid exposure of this

reference and limit the use of `metaActor` to (asynchronous) message sends. As with `meta` we change `Scanner` to recognize `metaActor` as a keyword and we modify `Parser` to allow only message sends.

The variable `metaActor` needs some additional computation. Message sends to `metaActor` have to be *translated* to message sends to the metalevel of the mirror's creator. We accomplish this by regarding message sends to `metaActor` as *meta message sends*. Meta message sends are processed differently in the interpreter,

- when evaluated in the interpreter, they operate in the metalevel of the actor, rather than the actor's behavior,
- they get priority over normal messages in the `inbox` of the actor.

A normal actor message is processed using the native function `execute`. For meta messages, we need a different function, which we name `executeMeta`. `executeMeta` changes the environment of the application to the metalevel of the actor and then executes the message as normal.

7.4 The familiy of metaobjects for actors

In chapter 6 we have identified the objects on which we reflect in the active layer of AmbientTalk. We distinguish three different kinds of metaobjects,

- the meta actor
- the behavior of an actor
- the metaobject for mailboxes and threads

Metaobjects for mailboxes and threads only differ in their interface, so we can reuse the factory pattern from the passive layer for metaobjects. The metalevel of the actor and its behavior require a different approach. Reflection on the behavior object of an actor is done using passive mirror methods and passive mirrors. This allows an actor to fully control the reflective facilities offered on its behavior. The metalevel of the actor is discussed below.

7.4.1 The meta of an actor

The meta of the actor is accessed by an active mirror using the reference `metActor`. This reference enables the mirror to invoke reflective operations on its creator. In chapter 6 we discussed that active mirrors invoke such operations using code blocks in the form of *functions* which they can send to the metalevel of the actor. For this, we include the function `metaActor#eval(codeblock)` in the `metaActor` reference. Code blocks executed with this method are evaluated in a context with access to the metalevel of the actor. This metalevel offers,

- `getMailbox(name, rights)`: returns a mirror on the mailbox with name `name` and the given `rights` as a policy,
- `getBehavior()` returns the behavior object,
- `getThread()` returns a mirror on the thread of the actor.

7.4.2 The meta of mailboxes

At the metalevel mailboxes offer the following operations,

- `getMessages()`: returns a table of mirrors to the messages from the mailbox,
- `addMessage(sender, message, arguments)`: adds the message to the mailbox,
- `removeMessage(message)`: removes the message from the mailbox.

7.4.3 The meta of threads

Metaobjects for threads have the following interface,

- `pause()`: pauses the current actor. It will only execute messages at the metalevel.
- `continue()`: continues the normal execution of the actor.
- `eval()`: will force the evaluation of a single message from the inbox of an actor, even if the actor's thread is paused.

7.5 The familiy of mirrors for actors

Active mirrors have an interface determined by their active mirror method and their construction has been explained in section 7.2. For mailboxes and threads, mirrors have an interface that depends on the rights policy given to them. We can reuse the technique of a mirror factory, which we used in the passive layer for objects, fields and methods. The only specification left for mirrors on mailboxes and threads is their interface which we have discussed in section 6.4.

7.6 Summary

We have implemented the native function `amirror` which can be used as method attribute to transform a declared method into an active mirror method in section 7.1. To support active mirror methods, we have extended the current function hierarchy in AmbientTalk with `AGAMirrorFun` in section 7.2. Upon invocation, active mirror methods create special actors called active mirrors. Active mirrors have a reference to the metalevel of their

creator named `metaActor`. We limited the use of `metaActor` to asynchronous message sends in section 7.3. In section 7.4 we elaborated on the different metafacilities for each element we reflect on in the active object layer. In section 7.5 we explained how we reused the techniques for reflection in the passive layer to implement the different kind of mirrors we can make for reflection in the active object layer.

Now that we have discussed the full implementation of our metaobject protocol, we are ready to evaluate it. We do this in chapter 8 and we draw our conclusion which we formulate in chapter 9.

Part IV

Conclusions And Future Work

Chapter 8

Evaluation Of The MOP

We now put our implementation to the test. We show the functionality of our metaobject protocol with some examples.

8.1 Reflection in the passive layer

In the passive layer we can now reflect on the structure of passive objects, as shown in this example,

```
helloObject:object{
 string: null;
 mirror.reflect()::{
  show(contents)::{display(contents.getValue())};
  stop()::{listener.unafter()};
  start()::{listener.after()};
  listener:=meta.after("string",show(meta.access(read));
 };
 init()::{reflect(); string:="Hello World"};
 init()
}
// --> "Hello World"
```

Upon construction, the object will call the method `init()` on itself. This triggers the construction of a passive mirror whom on creation attaches a listening mirror to the field `string`. Then the original object continues and sets its internal variable `string` to `Hello World`. This triggers the newly placed listener and will print out the value of `string`.

8.2 Reflection in the active layer

With the reflection in the active object layer, and the ability to passively attach listeners, we can create a logging mirror. First we construct a passive mirror method for the logging, then we construct an active mirror method to safely expose the passive mirror method to the active mirrors. Our code looks something like this,

```
anActor::actor({
  mirror.logg()::{
    on(action(name, args))::{
      meta.before("*", action(meta.call().method(read),
          meta.call().arguments(read)));
    }}
  ....
  amirror.logger(){
    remoteAction(name, args)::{
      thisActor#showAction(name,args)}
    showAction(name, args)::{display("invoking ",name, " with ", args)};
    metaActor#eval({
      meta.behavior().logg().on(remoteAction)});
  }});
```

8.3 A remote debugger

We can, quite easily, build a remote debugger. The debugger can pause the actor, and order a step by step execution of the messages in its inbox. An example code for such a program is given below,

```
anActor::actor({
  ... operations ...
  amirror.makeDebugger()::{
    pause()::{metaActor#eval({meta.thread().pause()})};
    continue()::{metaActor#eval({meta.thread().continue()})};
    step()::{metaActor#eval({meta.thread().eval()})};
    showNextStep()::{metaActor#eval({
     mbox: meta.getMailbox("inbox",meta.read);
         thisMirror.printout(mbox.getMessages()[0]);
})};
    printout(step)::{display(step.getMessage(),step.getArguments())}}
....
```

```
});
```

The debugger has an interface of four methods, `pause()`, `continue()`, `step()` and `showNextStep()`. With these operations the debugger can introspect every step, using `shotNextStep()` and control the thread execution using `pause()`, `continue()` and `step()`.

Our program examples have shown the basic functionality offered by our metaobject protocol. Based on that, and the design discussions from chapters 3 and 6 we can formulate our conclusions in the following chapter.

Chapter 9

Conclusions And Future Work

Based on the evaluation of our metaobject protocol in chapter 8 we can now formulate our conclusions in section 9.1. Then, in section 9.2 we discuss continuations on the design proposed in this thesis. We identify which parts the metaobject protocol presented did not cover but are of interest in a complete reflective protocol. Also, we discuss some research topics that may benefit from the approach presented in this thesis.

9.1 Conclusions

We have designed and implemented a metaobject protocol for AmbientTalk [9] based on design techniques presented in preceding research from chapter 2. In section 2.3 we discussed a design technique for reflective API's called *mirrors*, introduced by Bracha and Ungar in [2]. The benefits of mirrors can be summarized to three design principles, *encapsulation*, *stratification* and *ontological correspondence*. While these are desirable properties, as explained in section 2.3, the design proposed by Bracha and Ungar is in violation with the *extreme encapsulation* from De Meuter, introduced in [4]. In section 2.4, we elaborate on a proposal by Tanter from [27] to reconcile extreme encapsulation in mirror based reflection, using *mirror methods*. The proposal from Tanter sacrifices a part of *encapsulation* to restore *extreme encapsulation* in mirror based reflection for ChitChat [4]. In chapters 3 and 6 we transmuted the design technique proposed by Tanter to AmbientTalk. We proposed to reflect on the following language elements,

- passive objects,
- fields of passive objects,
- methods of passive objects,
- the behavior of passive objects, that is, by listening to language events on its members,
- active objects,

- mailboxes of active objects,
- the behavior object of active objects,
- the thread of active objects,
- messages between active objects

For each of the above language elements we have defined a metaobject which represents the element at the metalevel and offers operations on the element at the metalevel. To expose the metalevel in a controlled way for each of these metaobjects, we created mirrors. Passive objects, fields, methods, behavioral reflection, mailboxes, messages and threads were mirrored with a *passive mirror*. A passive mirror is a passive object with a reference to the metalevel of the object it mirrors. Active objects were mirrored with *active mirrors*. Active mirrors are active objects who keep a reference to the meta actor of the actor they mirror. The reference to the meta actor allows active mirrors to execute operations on the metalevel of the base actor. These operations on the metalevel of an actor are executed locally (cfr. section 6.3).
A crucial part of our design was to create mirrors in a controlled way, to preserve extreme encapsulation. For this we used the technique of *mirror methods*, proposed by Tanter in [27]. To incorporate the double object layer of AmbientTalk, we defined seperate kinds of mirror methods, one for passive objects named *passive mirror methods* and one for active objects named *active mirror methods*. We showed in sections 3.1 and 6.2 that following this technique, we preserved the *extreme encapsulation* principle. The designs described in chapters 3 and 6 were implemented and discussed in chapters 5 and 7 respectively. Finally we evaluated our implementation in chapter 8. We can conclude from our results that our metaobject protocol successfully offers reflection on the language elements stated above.

We now discuss the benefits of our proposed design. Our reflective API respects the stratification principle from mirrors. Stratification means that the reflective API can be loaded and unloaded dynamically depending on the application's requirements. In our design, no reflective system is loaded until an object (passive or active) creates a mirror (passive or active) to offer reflective facilities on itself. If an application uses no reflection, then in our design no object, passive or active, invokes a passive or active mirror method. Our reflective API can be neglected fully if an application does not require reflection. This strengthens the concept of *encapsulation*. Our design also upholds the ontological correspondence to some degree. Our design offers reflection on *most* structural elements of the language as stated above. In temporal correspondence, our metaobject protocol offers reflection on an object's (active or passive) behavior. One of the primary goals we have achieved is that our design upholds the extreme encapsulation principle. We have chosen to model *active mirrors* as *active objects*. The benefit of this approach is that mirrors have their own thread, own state and life. They can be moved to another ambient host. The mirror can store meta information collected from the base actor and offer controlled

reflection on this information to clients of the mirror, independently, or updates its information only when needed. This approach has some downsides as well, which we discuss below.

We now elaborate on the downsides of our design and implementation. The encapsulation principle of mirrors is not fully obeyed. To follow this principle would require us to remove reflection completely from the elements on which we reflect, which is in direct violation with the *extreme encapsulation* principle. We have opted to sacrifice a part of *encapsulation* in favor of *extreme encapsulation*. Our metaobject protocol offers no reflective API for tables, nor the virtual machine. The structural correspondence is thus incomplete. On the other hand, our design does offer reflection on *most* structural elements of our language as stated above. The behavioral correspondence is not complete either. We have no behavioral reflection for events on threads, tables or the virtual machine. There is also no reification of ambient events such as an actor moving from one ambient host to another. Another downside, and probably the most notable, comes from choosing active mirrors to be active objects. Active mirrors have their own evaluation thread and must communicate with their creator asynchronously. This means that there is a time discrepancy between a mirror and its creator. When pausing an actor's thread through its mirror, the response between the mirror and the actor is not immediate. Therefor our active mirrors are not good candidates for reifying the thread of the actor. A solution to this problem would be to implement active mirrors inside the actor and make the active mirrors and actors share the same thread. This, on the other hand has another downside. We must change the object model of AmbientTalk to some extent. Apart from the reified thread of an actor, there is little benefit in making a mirror share the thread of its creator. Clients communicate asynchronously with the mirror, which makes remote control over the thread asynchronous as well. Furthermore, active mirrors sharing the thread of their creator can not move independently from their creator. In section 9.2.2 we discuss reflection on the virtual machine. Within the context of reflection on a virtual machine, it is better to have active mirrors live independently in the ambient environment.

9.2 Future work

We have completed our implementation and drawn our conclusion. Before we turn the lights off, we first shed our light on some topics for future work or research that can build on top of the implementation presented in this thesis.

9.2.1 More reflection on AmbientTalk

The metaobject protocol in this thesis offers structural and behavioral reflection of passive objects, their fields, their methods, on active objects, their threads, their mailboxes and their behaviors. However, there are other language elements and constructs in AmbientTalk which have not been included in the reflection API presented in this dissertation such as tables. An extension of the metaobject protocol could also provide more low-level facilities

such as the following,

- A *statistics model* reifies technical information on an object, such as the amount of resources it requires or its state in the virtual machine. Especially for active objects, this may be an interesting feature.
- A *distributed environment model* reifies the actual location of an object in the ambient environment.
- A *migration model* describes how objects may migrate to other hosts. In case of passive objects, the host may refer to both virtual machines or active objects.
- The *lookup strategy* of fields and methods in the object could be reified in the metalevel. An object could use this to change its lookup behavior (i.e. forbid lookups in the parent). In section 5.5 we implemented a more flexible lookup strategy than the native behavior. Offering control over such lookup strategies allows an object to choose one of the strategies we have implemented (or other ones) as its default lookup strategy.

There a lot more events one could reflect on. We not claim this list is complete. The list stated above are but a selection of the topics we considered interesting.

9.2.2 A reflective virtual machine

An interesting topic for future work is reflection on the virtual machine. The current AmbientTalk interpreter abstracts hardware details such as the type of machine and the network protocols used by the virtual machine. It has been noted in [18] that applications may want to access this information inside mobile networks because of the heterogeneity of software and hardware aspects in such networks. For example an application may wish to notify the owner of an ambient device of an event. Depending on the hardware characteristics the application may choose to notify the user using a visual notification or a sonic notification.

But we can go a step further. Imagine we have an *extensible virtual machine* where we can extend the virtual machine with a form of plugins. For example, such a plugin may be a controller for additional hardware such as a WiFi or bluetooth card. The plugin is written in the native code of the virtual machine. Still, we need to expose these new facilities in the AmbientTalk layer so that actors may use the new plugin. We need a mechanism to bring plugin functionality in AmbientTalk. For this we propose a special kind of actors, named *virtual machine actors* (VM actors). VM actors are written in AmbientTalk and bound to a specific virtual machine (or plugin).They can offer services from the plugin in the virtual machine. In our example of a network card, the new communication service can he exposed in a VM actor named `Comm` which has control over the WiFi and Bluetooth card at its metalevel. It allows actors to communicate with the ambient environment using either the Bluetooth or WiFi network. At its metalevel, it has facilities that give direct

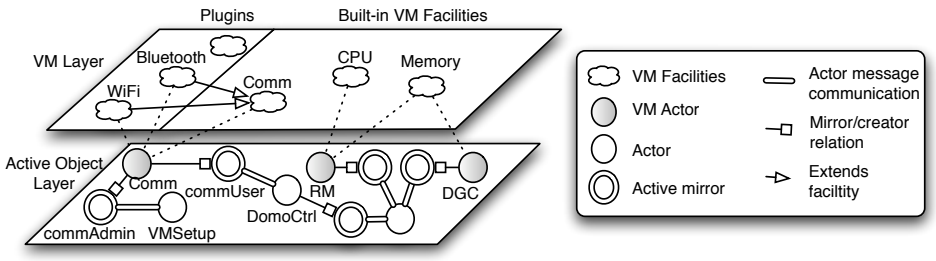

Figure 9.1: Exposing VM facilities with VM actors residing at the active object layer.

control over the corresponding hardware. Of course, not all actors should have access to such metalevel facilities. This is where our proposed design comes into play. Using active mirror methods in the VM actors, the virtual machine has selective control over which metafacilities are exposed. In the example of our `Comm` actor, an active mirror may offer limited control over the network for normal actors (e.g. a `CommUser`), while a virtual machine configuration actor (e.g. `VMSetup`) can have complete access over the plugin. This idea is represented in figure 9.1.

We list some of the typical metafacilities of virtual machines we could expose safely using VM actors and the mirror method-based reflection we proposed in this thesis,

- Hardware/Software profiling: this service reifies all facilities a VM has to its disposal (hardware or software). Using reflection, an actor may introspect or control the facilities a VM offers.

- Network profiling: this service reifies the identity, position etc. of the VM in the network.

- Communication layer: this resembles the comm service we used as an example above. It may allows control over the connection type (IPX, TCP/IP , ...) or network protocols like Bluetooth and WiFi.

- Distributed garbage collection: this facility offers actors control over the memory reclamation of actors.

- Resource management: a service to control the amount of memory and CPU time used by an actor.

- Discovery protocol: the discovery protocol is currently abstracted from actors. We could reify this at the metalevel of the VM and offer actors control over it. For example an actor may be interested in the frequency with which its required and provided services are broadcasted. He may increase this frequency for urgent situations or lower the frequency to, for example, save energy.

- Message delivery system: this system is also abstracted from the actors. Reification of this service would allow an actor to chose a suitable delivery policy, ensure quality of service, etc.

Not all facilities of the virtual machine are comprised in the list above. We investigated these topics as a point of interest, not with the ambition to offer a complete classification of a VM's facilities. Readers interested in a more detailed discussion on the cooperation between our design and reflection on the virtual machine are referred to [23].

Bibliography

[1] Gul Agha. *ACTORS: a model of concurrent computation in distributed systems*. The MIT Press: Cambridge, MA, 1986.

[2] Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2004)*, pages 331–344, Vancouver, British Columbia, Canada, October 2004. ACM Press. ACM SIGPLAN Notices, 39(11).

[3] Wolfgang De Meuter. Agora: The scheme of object-orientation, or, the simplest MOP in the world. In Ivan Moore, James Noble, and Antero Taivalsaari, editors, *Prototype-based Programming: Concepts, Languages and Applications*, pages 247–272. Springer-Verlag, 1999.

[4] Wolfgang De Meuter. *Move Considered Harmful: A Language-Design Approach to Distribution and Mobility in Open Networks*. PhD thesis, Vrije Universiteit Brussel, Belgium, 2004.

[5] Wolfgang De Meuter, Tom Mens, and Patrick Steyaert. Agora: reintroducing safety in prototype-based languages. In *ECOOP'96 Workshop on Prototype-Based Languages*, 1996.

[6] Wolfgang De Meuter, Éric Tanter, Stijn Mostinckx, Tom Van Cutsem, and Jessie Dedecker. Flexible object encapsulation for ambient-oriented programming. In *ACM Dynamic Language Symposium (DLS 2005)*, San Diego, CA, USA, October 2005.

[7] Jessie Dedecker, Stijn Mostinckx, Tom Van Cutsem, Wolfgang De Meuter, and Theo D'Hondt. Ambient-oriented programming. In *OOPSLA 2005 Onward! Track*, October 2005.

[8] Jessie Dedecker and Werner Van Belle. Actors for mobile ad-hoc networks. In L. Yang, M. Guo, J. Gao, and N. Jha, editors, *Embedded and Ubiquitous Computing*, volume 3207 of *Lecture Notes in Computer Science*, pages 482–494. Springer-Verlag, August 2004.

[9] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-oriented programming in AmbientTalk. In Dave Thomas, editor, *Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP 2006)*, Lecture Notes in Computer Science, Nantes, France, July 2006. Springer-Verlag. To Appear.

[10] Jim des Rivières and Brian C. Smith. The implementation of procedurally reflective languages. In *Proceedings of the Annual ACM Symposium on Lisp and Functional Programming*, pages 331–347, August 1984.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, October 1994.

[12] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[13] L. Gong. Jxta for j2me: extending the reach of wireless with jxta technology. Technical report, SUN Microsystems, 2002.

[14] J. Gosling and B. Steele. *The Java Language Specification, Second Edition, the Java Series*. Sun Microsystems, Inc, 2000.

[15] ISTAG. Ambient intelligence: from vision to reality. http://www.cordis.lu/ist/istag.htm, September 2003.

[16] A Kaminsky and H.-P. Bischof. Many-to-many invocation: A new object oriented paradigm for ad hoc collaborative systems. In *17th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA 2002)*, 2002.

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

[18] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, 2002.

[19] Pattie Maes. *Computional reflection*. PhD thesis, Artificial intelligence laboratory, Vrije Universiteit, Brussels, Belgium, 1987.

[20] Pierre Martin. A reflective approach to building extensible distributed actor-based languages. Master's thesis, Vrije Universiteit Brussel, 2006.

[21] Fred Rivard. Smalltalk: a reflective language. In Gregor Kiczales, editor, *Proceedings of the 1st International Conference on Metalevel Architectures and Reflection (Reflection 96)*, pages 21–38, San Francisco, CA, USA, April 1996.

[22] Brian C. Smith. Reflection and semantics in a procedural language. Technical Report 272, MIT Laboratory of Computer Science, 1982.

[23] Dieter Standaert, Éric Tanter, and Tom Van Cutsem. Design of a multi-level reflective architecture for ambient actors. In *ECOOP Workshop on Object Technology for Ambient Intelligence*, 2006.

[24] L. A. Stein, H. Lieberman, and D. Ungar. A shared view of sharing: The Treaty of Orlando. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 31–48. ACM Press/Addison-Wesley, Reading (MA), USA, 1989.

[25] Patrick Steyaert. *Open Design of Object-Oriented Languages – A Foundation for Specializable Reflective Language Frameworks*. PhD thesis, Vrije Universiteit Brussels, Belgium, 1994.

[26] SUN Microsystems. *The Java Language Specification*, 1996.

[27] Éric Tanter. Mirror methods — reconciling reflection and extreme encapsulation. In *ECOOP Workshop on Object Technology for Ambient Intelligence*, July 2005.

[28] David Ungar and Randall B. Smith. Self: The power of simplicity. In Norman Meyrowitz, editor, *Proceedings of the 2nd International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 87)*, pages 227–241, Orlando, Florida, USA, October 1987. ACM Press. ACM SIGPLAN Notices, 22(12).

[29] Tom Van Cutsem, Jessie Dedecker, Stijn Mostinckx, and Wolfgang De Meuter. A meta-level architecture for ambient-aware objects. In *ECOOP Workshop on Object Technology for Ambient Intelligence*, July 2005.

[30] John von Neumann. *The Computer and the Brain*. Yale University Press, June 1958.