Vrije Universiteit Brussel
Faculty of Sciences
In Collaboration with École des Mines de Nantes
2001-2002

Applying Prototype-Based Programming Techniques for the Organization of Mobile Systems

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

By: Boris Mejías

Promotor: Prof. Theo D'Hondt (VUB)
Advisor: Wolfgang De Meuter (VUB)

Abstract

Agents succeeded as mobile entities that properly solved problems on mobile systems. They can move around the network in a transparent way, and communicate with other agents by sending messages. But the relationship between agents does not have a strong structure, and big multi agent systems become hard to maintain and to understand. The primary aim of this work is to provide a way to organize agents in mobile systems.

Objects help us to have a structured way of programming, but if we want objects moving around different address spaces, we need them to be self-supporting. When objects depends on a class, we also need to move the class everywhere the object goes, and to have a sophisticated system for type checking. We decided to use Prototype-Based Programming, a paradigm to program without classes, which has a proper way of reasoning about sharing, an important property for our purpose to organize the system.

We extend Borg, a language to implement autonomous agents in a wide area network. The extension provides methods to create prototypes that can move and be extended through the network. The objects and their extensions use distributed delegation to share behavior and state, having a clear hierarchy and structure, helping us to maintain and evolve the systems.

Keywords: Prototypes, delegation, sharing, distribution, distributed delegation, organization, mixin-methods, network-mixin-methods, agents, mobile agents, Borg.

Acknowledgments

First of all I want to thank Wolfgang De Meuter, for all the advice, good ideas, time, music sharing, patience to explain me everything, and all the things I learned during this thesis period. Sincerely, Thanks Wolf!

I would like to thank Prof. Theo D'Hondt and people at Prog Lab of the Vrije Universiteit Brussel, for the support, equipment and the good environment. Thank to Johan Fabry for being always available helping me in the distributed topics and Borg. I want to specially thank Isabel Michiels for helping us in living in Brussel and coordinate everything related with the EMOOSE program, for the support and the friendship.

I also want to thank all the EMOOSE students for the nice time we shared during the master. Thanks to Michaël Vernaillen for helping us to arrive to Brussel and during the first week. Specially thanks to Sebastián González, for the support and all the discussions we had about prototypes. I would also like to thank Eduardo Miranda for all the nice time in Nantes, and the support from the distance, and Rhodrigo Meza for printing this thesis at Nantes.

I want to thank and dedicate my thesis to my parents, Jorge and Mary, and to my sister Mónica, for all the emotional background and the distributed support. Thanks a lot!

I also want to thank my friends, specially Saartje for supporting me and help me with my English.

Finally, I want to thank the organizers of the EMOOSE, Annya Romanczuck, Theo D'Hondt and Jacques Noyé, for make this program possible.

Contents

1 Introduction **6**
- 1.1 Motivation . . . 6
- 1.2 How to read this thesis . . . 8

2 Prototype-Based Programming **10**
- 2.1 Some philosophical and historical observations . . . 10
- 2.2 Classes vs. Prototypes . . . 13
  - 2.2.1 Creating a new object . . . 13
  - 2.2.2 Sharing behavior . . . 13
  - 2.2.3 Sharing state . . . 14
  - 2.2.4 The “self” variable . . . 14
- 2.3 Advantages and Disadvantages . . . 16
- 2.4 Objects with extensions . . . 17
  - 2.4.1 Split Objects . . . 17
  - 2.4.2 Agora . . . 19
- 2.5 Summary . . . 21

3 Distributed Programming and Mobile Systems **23**
- 3.1 Taking distribution seriously . . . 23
  - 3.1.1 Latency and Memory Access . . . 24
  - 3.1.2 Partial Failure and Concurrency . . . 24
- 3.2 Java RMI and CORBA . . . 25
  - 3.2.1 Java RMI . . . 26
  - 3.2.2 CORBA . . . 29
- 3.3 Obliq . . . 31
  - 3.3.1 Basic operations . . . 32
  - 3.3.2 Other features . . . 33
  - 3.3.3 Object Migration . . . 35
- 3.4 Borg: Agents and Mobility . . . 36
  - 3.4.1 Borg architecture . . . 36
  - 3.4.2 Programming in Borg . . . 37

3.5 Summary 41

4 Applying Distributed Delegation **42**
4.1 Extending a language 42
4.1.1 Macros and Procedures 42
4.1.2 Macros in Pico 44
4.1.3 Using Meta Programming 45
4.2 First Analysis 47
4.2.1 Delegation with Local and Distributed Mixin Methods . . 48
4.2.2 Sharing or not address space 50
4.3 P-Borg 51
4.3.1 Only Network Mixin Method 51
4.3.2 Implementing Distributed Delegation 52
4.3.3 The *us* and *yield* pseudo variables 55
4.3.4 Sending messages to the objects 57
4.3.5 Cloning 58
4.4 Summary 58

5 Validation **61**
5.1 The Role Object Pattern 61
5.1.1 Intent and motivation 62
5.1.2 Structure 64
5.1.3 Implementation 66
5.2 Chat 67
5.2.1 Java RMI 68
5.2.2 Borg 70
5.2.3 P-Borg 72
5.2.4 Using Mobility 76
5.3 A simple database 77
5.4 Summary 81

6 Conclusion and Future Work **82**
6.1 Conclusions 82
6.2 Future Work 83

A P-Borg Implementation **88**

List of Figures

2.1 The “self” variable. 15
2.2 Joe represented as a split object. 18

3.1 Java RMI Components at runtime. 26
3.2 Class and Interface hierarchy. 27
3.3 The OMA architecture. 30
3.4 The CORBA invocation architecture. 31

4.1 The difference between *clone* and *clone2agent*. 51
4.2 Distributed Split Objects. 52
4.3 The definitive model. 53
4.4 Sending ”setr” to a netcircle. 54
4.5 Sending ”setx” to a netcircle. 55
4.6 The “us” variable. 56
4.7 The “yield” variable. 56
4.8 The three kind of sending message to an object. 58
4.9 The cloning observation. 58

5.1 Customer hierarchy in a banking environment 63
5.2 An object diagram of the Role Object Pattern 63
5.3 The Customer object and its extensions in a banking environment 64
5.4 Structure diagram of the Role Object Pattern 65
5.5 Structure of the prototype-based approach 65
5.6 The Java RMI chat room. 68
5.7 Components of the Java RMI chat room at Run Time. 69
5.8 The Borg chat room. 73
5.9 The structure of P-Borg chat room. 74
5.10 The P-Borg chat room. 76
5.11 Data Base. 80

List of Tables

2.1 Comparison of the Two Paradigms 14

4.1 Elements of the table of a function in Borg 46
4.2 Elements of the table of an object in Borg 47
4.3 Methods of the extension of Borg 49
4.4 P-Borg grammar 60

Chapter 1

Introduction

A mobile multi agent system is one of the best approaches to apply the advantages and virtues of Mobile Computing. Agents can move around the network in a transparent way, and communicate with other agents by message sending. But multi agent systems have a lack of structure and organization in the relationship between agents, and big programs in this area become hard to maintain and to understand.

Objects are good to structure programming via encapsulation, and we claim they can be useful to organize mobile systems. But first we must ensure that we will use self-supported objects, because in other case it will be more complex to deal with mobility. If we use a class-based approach in mobile computing, as object depends on a class we must move the class wherever the object goes, needing complicated type-checking mechanisms to ensure correctness. For this reason we decided to used a prototype-based approach, and also because prototypes are good at reasoning about sharing, a property largely required in distributed systems with common data resources.

The primary aim of this work is to provide a way of organizing mobile systems. Our small contribution toward this aim will be focus in two goals: (1) Produce a conceptual language to study how prototype-based concepts can help us in order to organize mobile systems, and (2) Find uses, applications and advantages of distributed and mobile prototypes to validate our concepts.

1.1 Motivation

With the popularity gained by wireless connections in portable devices such as laptops, mobile phones and PDA's, Mobile Computing is becoming an important research area, having in mobile multi agent systems one of the best approaches to show how powerful and useful can be. The basic concepts largely studied in Dis-

tributed Computing become fundamental to build a good architecture to support multi agent systems. Problems such as fault tolerance, balancing and replication, take big part of the attention in the building of this kind of architecture, but from the organizational point of view, they are a little bit simplistic in their designs. In current day systems, agents know about each other through the "acquaintance" relationship, and this relationship is the only one currently used to structure such systems. As a consequence, it becomes difficult to understand why an agent belongs to a location, or which agents are allowed to see other particular agents.

Objects help us on having well structured programs, but if our desire is to have objects moving around different address spaces, we need them to be self-supported. When objects depends on a class, we also need to move the class everywhere the object goes, and we have to implement sophisticated systems only for type checking, like skeletons and stubs in Java RMI, or wrapped objects in middlewares such as CORBA and Voyager. We decided to use Prototype-Based Programming, a paradigm to program without classes, and where objects support themselves, making mobility easier to handle. This is the first reason to choose prototypes.

The dispute between classes and prototypes has a long history in philosophy, starting with Plato and his definition of the real world as instances of the world of ideas, continuing with Aristotle and his attempt to classify the entities of the world to its small details. But finally, philosophers concluded in a prototypical approach to conceive a representation of the world, based on the concepts presented by Wittgenstein and Rosh in the last century. In computer science, meanwhile classes have practically monopolized the object-oriented programming with successful languages such as SmallTalk, C++ and the popular Java, prototypes were popularized at OOPSLA'96 by Henry Lieberman, presenting an alternative way of sharing.

Prototypes are good at reasoning about sharing. Theories like Split Objects help us on defining structures to organize parent-child relationships to share behavior and properties between objects. As sharing common data resources is an important topic within distributed and mobile computing, we claim that prototypes is the best alternative to chose when the decision of using objects to model mobile systems is taken. The advantage on sharing is the second reason to choose prototypes instead of classes.

A common paradigm followed by many distributed applications is the client-server model, where a machine that plays the role of a server provides several services to different clients that are continuously making requests. The delegation relationship between prototypes appears as an analogy of this paradigm, where a "client" object delegates a service on its parent "server" object. We try to apply these concepts to distributed and mobile systems, trying to find advantages of mobility reducing network traffic.

Even when it is not our aim studying problems of the distribution scope such as partial failure, latency or balancing, we can not avoid them from our analysis. We strongly take them into account to design a realistic approach to be considered in future works. We include in our design concepts introduced in previous object-oriented distributed works such as Emerald and Obliq, but our work is mainly based on Borg.

Borg is a language designed to implement autonomous agents in a wide area network, presenting a very good architecture to support mobile computing. It provides transparency to the programmers in a distributed environment, and it has a very well design on strong and weak mobility, as one of its main features.

To get the first goal of this thesis, we are going to extend Borg, having a distributed prototype-based language as a result. We are going to name it P-Borg. With this conceptual language, we will be able to program objects with distributed extensions, moving around the network and delegating behavior and properties to other objects. The mobility and distributed features will be inherited from Borg, and the design will be inspired in the mixin methods of Agora, a prototype-based language with similarities with Split Objects.

For the second goal, the validation of our concepts, we are going to analyze and discuss three case studies, trying to have different points of analysis of the approach. First, we take a conceptual case presenting the Role Object Pattern, and how we can attack the problem from a prototypical point of view. As a second example, we compare three different implementations of a Chat Room, a well known problem in distributed computing, using Java RMI, Borg and P-Borg. Finally, we analyze a simple database to further exploit the features of our language.

We would like to remark that this work is about organization. We are not trying to extend the delegation mechanism of sharing to a distributed environment. We are trying to apply delegation and other concepts of the prototype-based theory, to help us on organizing agents in mobile systems.

1.2 How to read this thesis

In the present chapter we have introduced the motivation of the thesis, the problem we try to solve, and the goals of our work, giving a first idea about how we are going to attack the problem.

In chapter 2 we introduce the concepts of the prototype-based theory, including some philosophical and historical observations. We introduce the Split Objects theory, concluding the chapter with Agora, a prototype-base language that present similarities to the previous approach, but with some features that finally inspire our work.

The background in distributed and mobile computing is presented in chapter

3. We review the basic concepts on distribution, and some important approaches like Java RMI and Obliq. Finally, we focus on Borg, an experimental language that deals very well with mobile multi-agent systems, providing strong mobility and transparency to the programmers.

Our main contribution is presented in chapter 4. We start presenting a first analysis with important consequences for future works, and for the decisions we have taken to produce our definitive design of P-Borg. We describe the language, giving the semantics of the method and concepts implemented to help us in the aim of organizing mobile systems.

The validation of the concepts introduced in our approach is made in chapter 5, where three case studies are presented. First, we analyze the Role Object Pattern from a prototypical point of view. Then, we make a comparison between three implementations of a chat room, using Java RMI, Borg, and P-Borg, and finally, the design of a simple database, showing how the features of our extension can help to model and control a data resource distributely shared.

The dissertation finish in chapter 6 with the conclusions and future work.

Chapter 2

Prototype-Based Programming

Object-Oriented Programming can be seen from two different point of views. Class-based programming, the most popular one having in SmallTalk, C++ and Java its main languages. And Prototype-based programming, a way of programming without classes, and sometimes also named Object-based, because only objects exists in this approach

Prototypes were popularized at ECOOP '86 as an alternative object-oriented paradigm, when Henry Lieberman [Lie86] presented a different way of sharing behavior between objects, that later on motivates The Treaty of Orlando [SLU89], a Treaty two compare advantages and disadvantages between the way that class-based object and prototype-based object share behavior and properties. However, the basic concepts come from a long history of a philosophical discussion about the representation of abstractions.

We consider the philosophical arguments pretty relevant to reason about the advantages of prototypes, and in section 2.1 we present a brief introduction to that subject. In the rest of the chapter we focus on the comparison at the level of the languages concept, and finally we present an organized way of sharing thereby describing Split Objects and Agora.

2.1 Some philosophical and historical observations

Going out of the technical concepts that make the difference between class-based and prototype-based systems, we can find an old philosophical dispute concerning the representation of abstractions. In the early history, categories were used to model the objects of the real world; meanwhile prototypes appear in the last century as an alternative to solve the limitations of the classification.

Classes

Plato (428-347 BC) was the first philosopher in making an explicit distinction between *forms* – "ideal" description of things, abstraction – and *instances* of these forms. The world of ideas conceived by Plato is the analogue of a class model. Due to the explicit use of classes to represent similarities among objects, languages such as Smalltalk, C++, Java or Simula can be considered Platonic

Aristotle(384-322 BC), a student of Plato, did a research into biological classification of the world, trying to understand and organize it to its small details. A category was defined by *common properties* that a group of objects share. To define categories based on other categories, it is necessary that the new ones have at least the same properties ("genus") of the base one. Thus, the *essence* of a category is its *genus* combined with its *differentia*. The same concept is presented in the class-based programming, where a class is defined as an extension of a superclass (genus), plus additional behavior (differentia).

Aristotle realized that his model was not completely satisfying, because he found many objects presenting "accidental" properties. These accidents took those objects out of any classification, so it was not possible to represent the complete spectrum of objects using categories. Nowadays, while we are designing a class model, it is not difficult to encounter an object that does not match any of the classes of the model, and then necessarily a new class arises. Due to those "special" objects, the actual substance of concepts can be defined in terms of its *essence* and its *accidents*.

Prototypes

In the 20th century, *Ludwig Wittgenstein* observed that it is difficult to say in advance exactly what characteristics are essential for a concept. He defined what can now be seen as the origin of prototype-based programming: the notion of a *family resemblance*, where the meaning of the concepts can not be determined by definition. They can be defined only in terms of similarity and representative "prototypes".

A classical example is the concept of game. There are some games where there are no winners or losers, like ring-around-the-rosy. Sports like basketball, football or volleyball are games with different rules and number of players among other differences. Some games like board games involve luck, and some others involve skill, like chess. Poker is a game that involves luck and skill. All these examples belong to the category of games, even when they do not share a common behavior or definition. As we can see, its very difficult to define the essential characteristic that an object must has to belong to a class. The example illustrate that games are a sort of *family resemblance*.

Later on, in the mid-1970s, *Eleanor Rosh* observed and demonstrated that categories have best examples, what we call "prototypes". One of the principal arguments is the following: "If categories are defined only by properties that all members share, then no members should be better examples of the category than any other members". Then we can say that categories with best examples are not well defined.

In the real world there are categories with fuzzy boundaries implicating graduation and confusion. Thus, categories such as tall men, short distance or blue color have better examples. Even more, some people will consider a particular object ***p*** within a category, but some other people will not. Here comes again the argument that categories are not the best way to represent the knowledge of the objects that we want to model.

A typical argument in favor of prototypes is that people seem to be a lot better thinking first about examples, than conceiving abstract sets to represent a concept. This is the way that Henry Lieberman [Lie86] introduce the prototype-based paradigm with the example of the elephants. If we meet an elephant named Clyde, when we think about elephants we immediately think about Clyde, because is the only concrete example we have of an elephant. Then, if we meet another elephant, say named Fred, but with a different color than Clyde, we conceive the idea of Fread saying that is like Clyde but with a different color. Thus, the main idea behind prototype-based paradigm is going directly to the concrete example to represent objects of the real world, and then use cloning and adding new behavior to represent new knowledge of the model.

The limitations of the Aristotelian model of categorizing objects are inherited by the class-based programming. A class model needs a lot of iterations before being good enough to represent the real model. But considering the fact that there is no "optimum" class hierarchy, and the "perfect" design does not exist, the offering of the class-based languages of *being good enough*, satisfies enormously the requirements of software developers, being largely selected as the option to implement the solution.

One of the problems of the prototype-based paradigm is that its languages are not well developed yet, making them less famous and less attractive.

Summarizing, we can find the conceptual bases of the class-based programming in the ideas of Plato, and principally in the work of Aristotle. In the other hand, the prototype-based programming has its inspiration in the works of Wittgenstein and Rosch. There are other philosophers that have worked on these topics, but we consider these as the most important ones. More details can be found in [Tai96].

The technical concepts and the discussion about advantages and disadvantages between these two paradigms are presented deeper in section 2.2 and 2.3. The aim

of this historical introduction is not to bore the reader, nor to take him out of the subject. What we try to present here is that there is an important background that supports the prototype-base paradigm. Even though, the prototype-based languages still need to evolve a lot to be considered as a good alternative to develop software, and until now, the class-based concepts have dealt quite good modeling object-oriented system.

2.2 Classes vs. Prototypes

Now, going into the technical part, we will introduce the principal concepts of the prototype-based paradigm, identifying the differences with the class-based orientation. The main concept is that in this paradigm there are no classes, just objects, and each one of them is consider a prototype that can be cloned.

2.2.1 Creating a new object

As we know, in class-based languages, objects are the result of instantiating a class. Thus, an object can not exist without a class that defines its behavior and its way to represent its state. Furthermore, once an object is instantiated its behavior and state can not be extended anymore.

In a prototype-based language, objects support themselves and do not need a class to exist. An object can be created Ex Nihilo and-or by *cloning* an existing one. New behavior and additional state information can be added to the object after its creation. Thus, the clone will have at least the same behavior and state information of its *parent prototype*, plus its extension.

Depending on the language design, when the object is created Ex Nihilo, there are two possibilities: A new empty object, what is only useful when the language allows to modify dynamically its structure; or and object with initial information than can be passed as an argument.

To clone an object, we can also use two techniques. Deep copy, where the values of the attributes are also copied to the new object. Shallow copy, where only the attributes and the pointers of the values are copied. Most of the time the cloning take a shallow copy of the behavior, and a deep copy of the state.

2.2.2 Sharing behavior

Inheritance is the methodology used by in class-based languages to share behavior between objects. A class 'A' inherits the behavior of a class 'B' if 'A' is a subclass of 'B' - we say that 'B' is the superclass of 'A'. Then, all the instances object of class 'A' can access the behavior of instances of class 'B'.

In prototypes, the way of sharing behavior between objects is by *delegation*. When a message is sent to an object, and this one does not know how to handle it, it delegates the message to its list of prototypes. In this point we want to make clear the difference between message passing and delegation. In delegation, all the environment of the object is sent to the parent prototype, then, the method is evaluated in the context of the original receiver of the message. In message passing, the message will just be re-sent to the parent object, performing the message in the context of the parent.

We can make delegation implicit or explicit. When delegation is implicit, the interpreter will automatically delegate the message to the parent objects, following the parent-links. Sometimes it is possible to modify dynamically the parent link, but some others this is not allowed. The explicit delegation is completely controlled by the programmer, but keeping the difference with message passing.

2.2.3 Sharing state

The instances variables defined in a class are shared by all instances object of that class. Nevertheless, each instance object has a copy of these variables that can not be shared by any other object. Then, the state of the objects can not be shared.

To share state in prototypes, an object can delegate a message to set variable attribute to another object. If the parent object change the value of the variable, it will have an effect in the object that delegates the variable. Then, objects are able to share their states.

The table 2.1 presents a comparison of the two paradigms.

	Classes	**Prototypes**
Creating Objects	*new* (instance of a class)	Ex Nihilo and-or *clone*
Sharing Behavior	Inheritance (superclass)	Delegation (parent object)
Sharing State	No	Yes

Table 2.1: Comparison of the Two Paradigms

2.2.4 The “self” variable

One of the arguments used to compare the two paradigms is the expressive power of the languages. In [Lie86], Lieberman showed that prototypes can simulate class inheritance, but classes can not simulate delegation. To demonstrate that, he used the “self” problem of the classes but focused on instance objects, instead of using an approach at the level of classes, as it is done in [Ste87]. In that approach, they

demonstrate that both paradigm can simulate each other. Then, both are equivalent in the expressive power.

In this report, we will not go deeper into the demonstration of expressive power, but we will review how the "self" variable works. The difference lay on where the "self" variable is bound. In classes, the "self" is bound on the object where the message is defined, meanwhile in prototypes, the "self" returns to the original receiver of the message, executing the method in the context of the objects that inherit the method. Figure 2.1 depicts the difference.

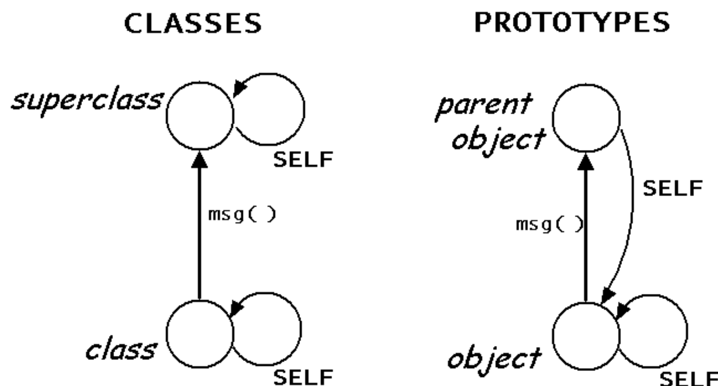

Figure 2.1: The "self" variable.

Let us suppose we have an object **b** that belongs to the class **classB**, that inherits from **classA**. The definition of the class is as follows

```
classA {
     variable x;
     method get(){ print(x); };
     method m(){ self.get(); };
   }

   classB extends classA{
     variable y;
     method get(){ print(x + y);};
   }
```

If we send the message **m** to the object **b**, as it inherits the method from **classA** it will be executed but in the context of the supper class. Means that the method **get** that is invoke from method **m** corresponds to the one defined in **classA**, printing only the variable x.

Now if we have two objects **a** and **b**, with a similar definition as the example with the classes, and with **b** delegating to **a** as follows.

```
a{ variable x;
   method get(){ print(x); };
   method m(){ self.get(); };}

b{ parentlink a;
   method get(){ print(x + y); };}
```

If we send the message **m** to the object **b**, it will be delegate to the object **a**, and it will be executed in the context of **b**. Then, as the self variable remain being the **b** object, the method **get** that is invoke from method **m** corresponds to the one defined in object **b**, printing x + y

2.3 Advantages and Disadvantages

In addition to the conceptual advantages of the prototypes over classes presented in 2.1, here we go more into the comparison between these two paradigms at the level of languages concepts. The reader might disagree with some of the points listed in this section, what is understandable since the discussion is still open and lot of research can be made in this area.

Advantages

- Objects do not need a class to exist. They support themselves. Sometimes there are systems with classes that just have one instance of them, e.g., in Smalltalk, the objects *true* and *false* need a class *True* and a class *False* respectively to exist, being the only instances of each class.
- Objects can share the state. We already had the discussion in section 2.2
- As a shallow copy of the behavior is made in a cloning, prototypes save memory space.
- Prototypes can change the behavior dynamically, and extend it.
- Prototypes deal better with software evolution, because the model can be extended instead of change it, moving code to superclasses, creating new abstract classes, and using other techniques to evolve class-based systems.
- The dynamicity of prototypes helps to deal with mobility in distributed computing, and principally with agent programming. This will be clearer in chapter 3

- Prototypes are simpler. People deal better in the generation of models having first concrete examples than conceiving abstractions.

- As all the objects are in the same layer, prototypes are better for doing reflection, because they do not have the complicated metacircularity of classes and metaclasses.

Disadvantages

- The delegation methodology save memory space but implies a reduction of the speed of the program, because of the chain that is built delegating over a prototype that also delegates.

- Some prototype-based languages do not make the distinction between inheritors and clients. Then, as it is explain in [Ded01], all the objects can breach the encapsulation, what is one of the merits of the object-oriented paradigm.

- Some concepts are not easy to be expressed with a prototypical example, for instance an integer. Other kind of concepts where the boundaries are well defined seems to be easier to be modeled with classes.

- The dynamism of prototypes makes the checking of correctness at compile time harder. This is easier with the static typing system of classes, adding safety to programs.

Maybe the stronger advantages of prototypes lay on the basic concepts of the philosophical area, on the simplicity of its concepts, and on its dynamism. Even though, we are not trying to say that prototypes are "the" solution for our problems. As we can note, the disadvantages that presents make us realize that sometimes is better to use classes to design our solution.

Abstraction is powerful and classes are useful, but they present several limitations that can be solved in many cases by using prototypes. Finally, the characteristic of the problem domain will lead us to decide between the two paradigms to design our solution.

2.4 Objects with extensions

2.4.1 Split Objects

Split objects were designed to clarify and characterize the kind of sharing achieved by delegation. The theory concludes that delegation can be used to achieve a

per-viewpoint representation of a single entity of the real world. A viewpoint of an entity can be seen for instance, as a role that a person can play in a certain moment. Taking the example of a person named Joe, a viewpoint of Joe will be Joe as a sportsman. Then, if we define the object JoeSporstman to represent this viewpoint, it will share common properties with the object Joe. In this case delegation is clearly a good mechanism to share behavior or state between objects.

The basic model for split objects consists in the representation of an entity of the real world as an object with different viewpoints represented by pieces. Pieces are organized within an object in a delegation hierarchy. Note that a split object represents a single entity of the real world, and its pieces are just viewpoints of the same single entity. Thus, **pieces do not have an object status, whereas split objects do**. A representation of Joe as an split object is depicted in figure 2.2 containing five pieces in a delegation hierarchy.

Note that delegation in split object is used **inside** the objects to express property sharing between different viewpoints. We consider this kind of delegation a good methodology of organization, and we are going to use it in our conceptual language.

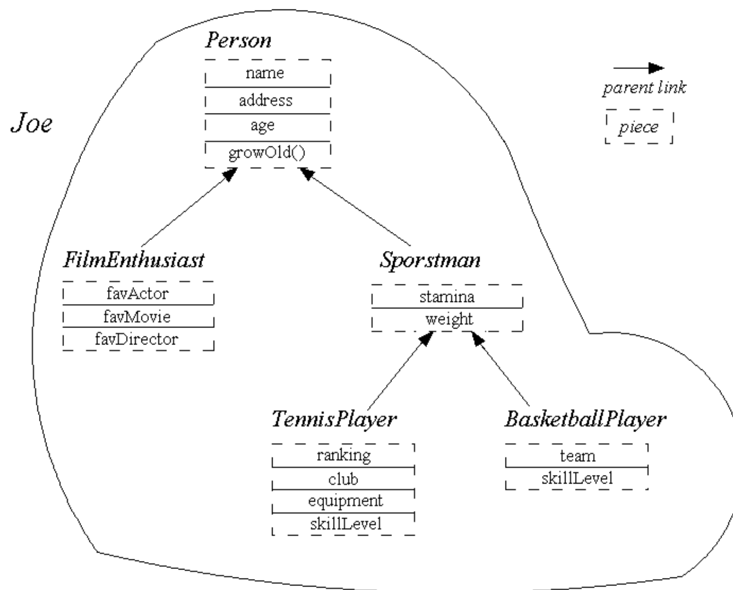

Figure 2.2: Joe represented as a split object.

Naming and accessing

Split objects are first class entities of the model. Then, they are directly accessible by its name. In the case of pieces, as they are not objects of the model, they are

only accessible within the objects by its name. They can not be accessed from out of the hierarchy of delegation.

Creation and cloning

All the objects are created by cloning mechanism. To create an object ex-nihilo, one may clone the *empty split object* predefined by the language, and then add the require pieces. Cloning an existing split object results on a new split object with the same set of pieces and properties. The entire pieces hierarchy is cloned using deep copy, meanwhile every piece is clone by using shallow copy.

Message sending

The way to communicate split objects is by message sending. As variables and methods are owned by the pieces, one may identify the piece that we want to refer. Then, the method-lookup will start in the piece identified by the sender, and if the method or the variable is not found, it will delegate the message to the parent link.

In the following section we will review Agora, a prototype-based language that use a similar approach having objects with extensions. The difference lay on the possibility in Agora of accessing the extensions as individual objects.

2.4.2 Agora

Among prototype-based languages, SELF [US87] is maybe the most emblematic one, but there are some others such as NewtonScript [SLS+94], Cecil [Cha92], Obliq [BC95, Car94], Pic% [D'H] and Agora [Meu98a, Meu98b] where important research has been done. For the aim of this thesis we will focus on Obliq, presented in section 3.3. We will also review Pico [D'H] in section 4.1, which is a functional language where Pic% and Borg [Lab] are based on. In this section we want to introduce Agora, taking its mixin-methods as an inspiration to build our distributed inheritance system.

The Language

Instead of consider Agora as just one language, we shall say that it is a language family, due to the fact that Agora is a framework for building prototype-based languages. Now, we will just focus on the common features of the language.

Agora has a dynamic typing system and a syntax similar to Smalltalk. Only objects exist and the only thing you can do with them is sending messages. Like Self, Agora is *slot-based*, and has the distinction between local and public attributes. The *local* concept has no relation with distributed programming, it is

used to define attributes that are encapsulated into the object and are only available by itself. Meanwhile public attributes become the interface of the object. This is like private and public in Java.

Creation and Delegation

Respecting the basic concepts of the paradigm, objects are created Ex Nihilo or by cloning. To extend the behavior of the object, Agora provides mixin-methos, that will be explained bellow.

The language use *implicit* delegation. Thus, as the parent link is not available, the parent of an object cannot be modified once it is created.

Features

There are some interesting and useful features in Agora such us *Reifier Messages*. We can send ordinary messages to objects that perform as normal messages and reifier messages that are not necessarily evaluated, and that are sent to the abstract grammar tree, allowing the programmer to extends the language.

The meta object protocol is really simple, because it just consist on the send-message. To extend Agora is quite desirable to use reifier messages rather that expanding the MOP.

Agora has a reflective architecture that allows to move objects from the base level to the Agora language by sending them the message *up*. The message *down* take the objects in the other direction.

But the principal feature that we want to explore with more details is the mixin-methods.

Mixin-Methods

To modify the structure of an object at runtime, Agora provides mixin-methods that allow to override methods and variables, and to add new ones. The extension is performed when the mixin message is sent to the object.

It exist two kind of mixin-method, *functional* and *imperative* (or destructive). The functional, identified in Agora98 [Meu98b] by the keyboard *view*, extends the object adding a new frame where the new methods and variables are installed. The receiver of the message does not change, but it is extended. In the imperative, created by using the keyboard *mixin*, the receiver of the message destructively change. That is the difference between both.

We can define new mixin-method within previous one, obtaining extensions of the extensions of the object, creating a hierarchy of mixins.

For our distributed system we will focus on the functional mixins, looking for extending the objects through the net. The following code shows a simple example in Agora98 to extends a point into a circle, with the respective getters and setters. In chapter 4 we will do the same but in a distributed way.

```
point VARIABLE: [

        x VARIABLE: 0;
        y VARIABLE: 0;

        getx METHOD: x;
        gety METHOD: y;
        setx nux METHOD: x: nux;
        sety nuy METHOD: y: nuy;

        circle: aR VIEW:
            {   r VARIABLE: aR;

                getr METHOD: r;
                setr nur METHOD: r: nur}

    ];
```

Once the structure is defined, the way of creating objects, its extensions, and sending messages to them, it is shown in the following code:

```
aPoint VARIABLE: (Self Point) clone;
aPoint setx: 3;
aPoint sety: 5;

aCircle VARIABLE: aPoint circle: 7;
aCircle setx: 0;
aCircle setr: 9;
```

2.5 Summary

During this chapter we have presented the main concepts of the prototype-based paradigm. We started in section 2.1 with some philosophical and historical observation that help us to reason about the conceptual importance of this paradigm.

In section 2.2 and 2.3, we made an overview of the principal concepts of the prototype-based languages, making a comparison with the class-based languages,

listing the advantages and disadvantages, concluding that prototypes are not "the" solution for our problems, but it is a very good alternative to deal with the limitation of classes.

Finally we presented Agora and its mixin-methods, that will take relevance in chapter 4, when we present how we can delegate and extend an object in distributed computing.

Chapter 3

Distributed Programming and Mobile Systems

Important concepts such as latency and partial failure, have not been taken into account by several distributed approaches, causing that many applications built with these approaches fail. We consider, and we are definitively not the first ones, that distributed computing have to be taken seriously, because it is scenically different from local computing.

In section 3.2.2 we introduce CORBA [Sie00], and we explain why we dislike the vision of unified objects for distributed computing. In section 3.2.1 we briefly describe Java RMI [Mic], concluding that classes make distribution more complex, and such a approach is not good for mobile computing.

Among well designed languages for distributed computing such as Emerald [Hut96, NCRB^{+}87], Obliq [Car94] and VisualObliq [BC95], we describe Obliq in section 3.3, presenting its features and concepts that can be useful for our goal in this thesis.

Finally we describe Borg [Lab, BFVD00], a well designed experimental language that supports mobile multi-agent systems. Together with Borg, we describe the concepts on mobile computing that we are going to use to extend this language to generate P-Borg.

3.1 Taking distribution seriously

In this section we want to introduce the main concepts that make distributed computing different from local computing. Concepts that have to be taken into account when we want to apply a prototypical approach in a distributed and mobile system.

Considering objects as the entities of our programs, we consider local computing when all the objects share a single address space. We mean by distributed

computing objects in different address spaces on the same machine, and objects in different address spaces on different machines, and maybe with different architectures.

We are going to review the four main concepts that differentiate this two kind of computing, finding in the partial failure problem the most conceptual difference between both.

3.1.1 Latency and Memory Access

The most obvious difference between local and distributed computing, is the latency of a remote call compare with the latency of a local call. Even though, several systems ignore this difference arguing that in the near future this difference will be minimal without being necessary to consider it. But ignore the different of latency is to ignore one of the major design areas of an application. Using objects, it is important to design what objects can be remotely called, and what objects must clustered together. Moreover, objects in non-distributed environment are programmed considering that answers and the returning of values are immediately performed, what is not true in distributed systems.

Being the latency problem the most obvious difference, several systems get blind and consider it the only difference between local and distributed computing. But this is not the only and not a fundamental difference. In the following section we explain the problem of partial failure and concurrency, but memory access is another important point to take into account.

The difference accessing memory of a remote data resource and a local access is presented majority in pointers. Pointers in a local address space are not valid in another remote address space. To solve the problem we have two alternatives: We delegate the control of memory accesses by underlying system, or the programmer must be aware of the difference.

Unifying the programming model, making remote access behaves as local access; the total control of the memory access is done by the underlying mechanism. With this decision, programmers who start with this style of programming will never know about address-space-relative pointers. For pedagogical reasons, and also industrial training, it is better to make programmers aware about distribution. An aware programmer will not use pointers for cross-address space access

3.1.2 Partial Failure and Concurrency

Meanwhile latency and memory access are problems that could be masked satisfactorily, partial failure can not. The partial failure problem represents the most conceptual difference between local and distributed computing.

In local computing, a fail means either the entire system fails, or there is a central control (such us the operative system) that detects the fail and handle it. In distribution, one component (machine, network link) can fail while the other components continue running, and there is no common agent to detect the fail, nor a global state to determine where the error occurred. It is not even possible to distinguish when the failure occurred in a machine or in a network link.

As we can see, the partial failure problem is enough to conclude that distributed computing is different to design than local computing. Furthermore, there are similar arguments to make the distinction in the way of attack the problems in concurrency between local and distributed computing.

Because of the nature of distributed systems, components have to deal with concurrent method invocations. In local computing, we find in multi-threads the same situation. The problem is that in distribution, there is no single point of resource allocation, synchronization, or failure recovery, but in local computing there is. Moreover, distributed computing introduces truly asynchronous operation invocations, meanwhile in local computing programmers can deal deciding the sequences of concurrent method invocations.

3.2 Java RMI and CORBA

There are several approaches in class-based programming to implement distributed applications. In this section we present Java RMI, that provides to Java programmers to do Remote Method Invocation, as its acronym indicates. We also present CORBA, a very popular architecture provided by the Object Management Group (OMG) which aim is unify the objects in a distributed environment. As both approaches are very well known, we are just going to introduce them briefly, without going into details of their features and services.

We discard a class-based approach for our aim, because we consider class-based objects not good for mobile computing. Even though, we decided to included Java RMI as a point of comparison for our reasoning. In chapter 5, we compare three different implementations for a chat room, and one of them is implement with Java RMI.

Instead of give a detail presentation about what CORBA is, after a brief introduction we will focus on the reasoning why we do not think that unifying objects are good for our purpose of organizing mobile systems. The reasoning is based on the article [WWWK97].

3.2.1 Java RMI

One of the possibilities that Java provides to implement distributed applications is Java RMI, a mechanism with similarities with the Remote Procedure Calls (RPC), giving a high-level communication abstraction. RMI allows Java programmers to have control transfer between caller object and called object.

The general idea to provide communication between a server object with a client object, is the following: There is a name service called *Registry*. When a server object is created, the object publishes its reference on the *Registry*. Then, a client object can get the reference to the server from the *Registry*. To perform a remote method invocation from the client to the server, a *Stub* of the server is placed in the location of the client to receive the call. The message is sent to the *Skeleton* of the server throw the socket that communicate client with server. The *Skeleton* is placed on the server location, connecting the server with the socket. Then, the server receive the message to perform it. The structure of components at runtime is depicted in figure 3.1.

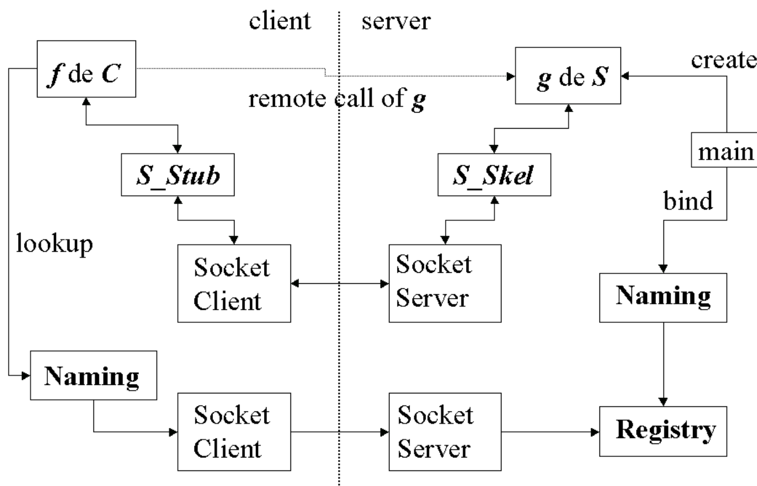

Figure 3.1: Java RMI Components at runtime.

Note that this sophisticated structure of *skeletons* and *stubs* is only designed to have the types of the objects right. We do not need such a structure to communicate two objects in a prototype-based approach. Moreover, classes need to respect an hierarchical model showed in figure 3.2. We can see that every class that implements objects using RMI, must extend the class `UnicastRemoteObject`, and implements its own interface. That interface always have to extends from the Java Interface named `Remote`.

Let us review a very small example using Java RMI. We have a server named

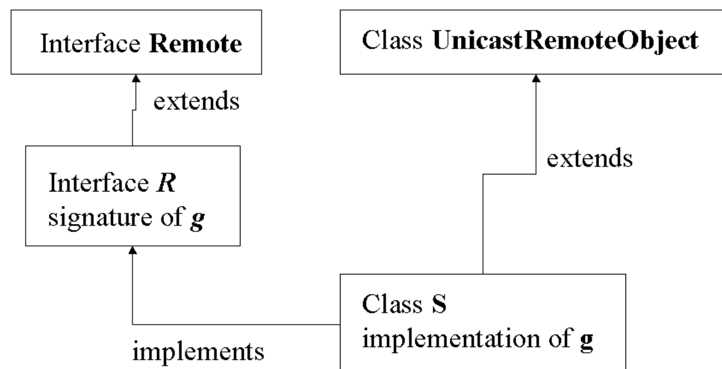

Figure 3.2: Class and Interface hierarchy.

`Calendar` providing a `date` for the clients. The interface will be as follows[1].

```
// file iCalendar.java
// specifies a date server interface

import java.rmi.* ;

public interface iCalendar
      extends Remote
{
   java.util.Date getDate ()
   throws RemoteException ;
}
```

Then, the implementation of this interface is the following. Note that there is a lot of bureaucracy in the code, in the real implementation is the method `getDate`. The `main` method does the publishing on the *Registry*. The *stub* and the *skeleton* are generated automatically with the RMI compiler `rmic`.

```
// file CalendarImpl.java
// the date server implementation

 import java.util.Date;
 import java.rmi.*;
 import java.rmi.registry.*;
```

[1]The example is taken from the course of Distributed Objects, EMOOSE, November 2001

```
 import java.rmi.server.*;

public class CalendarImpl
   extends UnicastRemoteObject
     implements iCalendar {

   public CalendarImpl()
   throws RemoteException {}

   public Date getDate ()
   throws RemoteException {
   return new Date ();
   }

public static void main(String args[]) {
  CalendarImpl cal;
  try {
    cal = new CalendarImpl();
    LocateRegistry.createRegistry(1099);
    Naming.bind("rmi:///CalendarImpl", cal);
    System.out.println("Ready !");
  }
  catch (Exception e) {
    e.printStackTrace();
  }
}
```

Now let us review the client object. The client does not know anything about the implementation of the `Calendar`, the only thing important for the client is the interface of the server. The reference to the server is get in the `main` , set in the variable `remoteCal`. Note that the object class `Date` provides the method `getTime`.

```
// file CalendarUser.java

 import java.util.Date;
 import java.rmi.*;

public class CalendarUser {
  // constructor
  public CalendarUser() {}

  public static void main(String args[])
```

```
  {
    long       t1=0,t2=0;
    Date       date;
    iCalendar  remoteCal;
    try {
      remoteCal = (iCalendar) Naming.lookup
      ("rmi://some.host.com/CalendarImpl");

      t1 = remoteCal.getDate().getTime();
      t2 = remoteCal.getDate().getTime();
    }

    catch (Exception e) {
      e.printStackTrace();
    }

    System.out.println
      ("This RMI call took "
       + (t2-t1) + " milliseconds");
  } // main

} // class CalendarUser
```

We already argued that classes are not good for mobile code, because objects depending on a class need to carry the class with the, everywhere they goes. Using Java RMI, we can see in the code and in the structure concept, that a complete system for type-checking must be implement. We prefer prototypes because they are simpler and self-supported. In chapter 5, we are going to compare a chat room implement with Java RMI, against an implementation with Borg, and with our extension of Borg.

3.2.2 CORBA

The Common Object Request Broker Architecture is designed to support distributed object-oriented programming with the aim of integrate diverse applications in a heterogeneous environment. We are not going to detail its structure, but we want to analyze why we consider the CORBA philosophy not good for distributed programming, in the sense that CORBA unify all the objects without making a clear distinction between local and distributed objects.

Let us briefly review the general concepts. The main architectural aspects are defined in the Object Management Architecture (OMA) depicted in figure 3.3.We can see that all the communications and services in CORBA are performed through a middleware, the Object Request Broker (ORB), supporting remote method invocations and making

the remote object references transparent. The middleware communicate the applications objects with the CORBA domain, facilities and services.

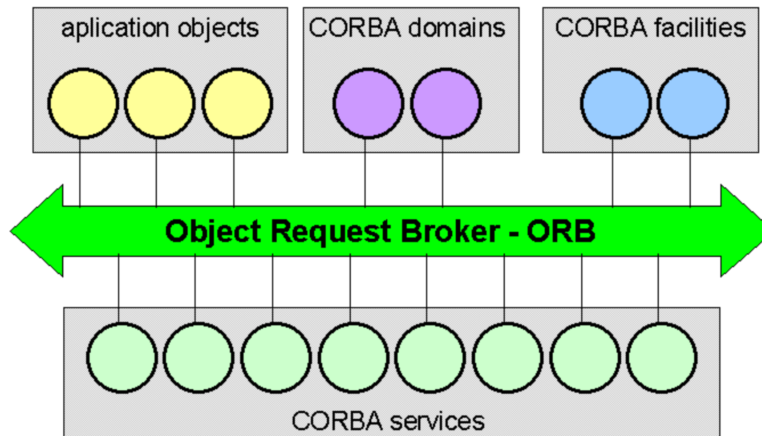

Figure 3.3: The OMA architecture.

The CORBA services include basic system-level services such as naming, event control. The facilities include high level functions, user interfacing, information management, etc. We mean by CORBA domains, functions in a specific domain such as telecommunication, finances, manufacturing, etc. We are going into detail of these services of CORBA, but is clear that its aim is unify the World Wide Web.

The ORB, locating the target object and marshalling parameters and result transparently manages the remote method invocation. It offers transparency of languages integrating, Java, C++, SmallTalk and others. The location of the objects is also transparent, and the object references are resolved by the ORB, unifying all the objects, and using the same technique as Java RMI, communicating objects though their stubs and skeletons. The invocation architecture is depicted in figure 3.4.

All the architecture is designed to have unified objects to integrate diverse application. What we do not like of CORBA, is the methodology that they promote to program distributed applications with all this concepts. The methodology of unified object can be described in three phases.

- Build as a local application: Write the application without worrying about where objects are located and how their communication is implemented. This is possible while having a correct interface between objects.
- Tune performance: Concretize object locations and communication methods. This could be made by several ways, using tools to analyze communication patterns between objects, or taking the decisions manually.
- Test with “real bullets”: We mean by real bullets failures such as machines going down, networks being partitioned, etc. Only the experience will be useful to determine how to test the system to debug it.

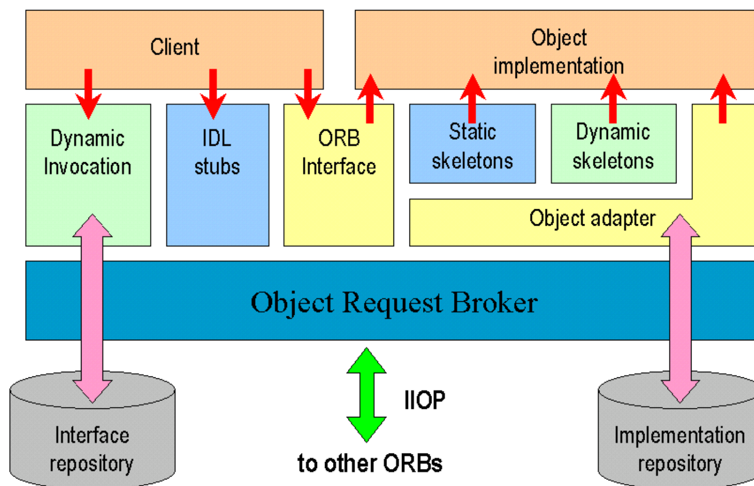

Figure 3.4: The CORBA invocation architecture.

Unfortunately these concepts male several applications fail because of the four concepts we already describe in section 3.1. ORB can mask successfully the differences on latency and memory access between local and distributed objects, but the problems with partial failure and concurrency are still unsolved, and if the programmers are not aware of these concepts, applications will not be able to handle these failures.

Unifying objects as local objects, partial failure will be fragile and non robust, because object can not rid of those failures. The other possibility is unifying objects considering them all as distributed objects. Then, all the forms of distributed indeterminacy are forced to be dealt with on all object invocations, making local computing as complex as distributed computing.

In the following sections we will present Obliq and Borg, two distributed languages that from our point of view, are better dealing with the basic concepts of distributed programming, and presenting better features to deal with mobility, compare with class-based approach in distribution like Java RMI.

3.3 Obliq

Obliq is defined as “lexically-scoped, untyped, interpreted language that supports distributed object-oriented computation”. It has important concepts that will be useful for our aim, having a distributed lexical scoping as the key mechanism to deal with distributed computing. Obliq structure distributed computations using objects.

Obliq objects are self-supported because they do not depend on a class, and then, can deal better with migration. They are defined as collections of named fields. As they are untyped, communication between them does not need complicated structure for type checking as in Java RMI. Objects are just known by *network references*, keeping the

communication transparent throw the net.

One of the important features is that every object is potentially and transparently a network object. An object become accessible either using a naming server, or as an argument or result of a remote method. What is missing in Obliq is a way of share behavior or state between objects. It can be considered as a prototype-based language, but is not delegation-based. The notion of a *super* it does not exist. All the methods and value fields are embedded in the object itself.

3.3.1 Basic operations

Objects have four basic operations apart from the creation. They are described bellow.

Selection (and Invocation)

There are two ways for selection and invocation of method, using parameters or not. In the following examples, the first one represent a selection of the field **x** of the object **a**, and the second one is the invocation of a method **x** supplying parameters.

```
a.x
a.x(b1, . . ., bn)
```

Updating (and Overriding)

The same operation is used to update a field or to override a method. It will depends on which kind of concepts is behind the variable we are updating.

```
a.x:b
```

In the example, if **x** is a field of the object **a**, it will be updated will the value of **b**. And if **x** is a method, it will be override by **b**. Note that we could mix method and fields, meaning that if **x** is a field and **b** is a method, **x** become a method, and if **x** is a method but **b**, **x** become a field. We will see in the following section that Borg, based on Pico, used the same semantics for the assignments.

Cloning

For cloning objects we have three possibilities. The first one is clone from one object, which is given as a parameter of the operator.

```
clone(a)
```

The clone operator can receive multiple parameters, then, we have two new options: we can clone from several objects, inheriting all its fields and methods. And, we can clone and add new fields that are not defined in other objects, as it is shown in the example when we clone from an object **a** giving new field in the second argument. An error will be generated if there are name conflict in the multiple cloning.

```
clone(a₁, . . . , aₙ)
clone(a, { . . . })
```

Redirection

Redirection is the replacement of fields with aliases. To make clear the difference with aliases, the following example describe how **x** is an alias **y** of **b**.

```
{ x => alias y of b end, . . .}
```

The redirection assigns aliases to existing fields of objects. The syntax is similar to updating, but the semantic is completely different. The following example means that the field or method **x** of **a** is redirected to the field or method **y** of **b**.

```
a.x := alias y of b end
```

To redirect every field and method of an object a_1 to every method and field of an object a_2, the following operation is provided by the language.

```
redirect a₁ to a₂ end
```

3.3.2 Other features

Apart from creation and the four basic operations already described, Obliq provide other interesting features. For instance, with the aim of safeguard the internal invariance of objects, Obliq provides a keyword to protect objects. Then, we could protect server objects from clients overriding methods. Objects are declared protected as follows.

```
protected, x₁ => a₁, . . ., xₙ => aₙ
```

To deal with concurrency problems, objects can be serialized to use the often techniques of serialization. Obliq also provides a way to install guards. In the following example we can see how to serialize objects, and to install a guard.

```
serialized, x₁ => a₁, . . ., xₙ => aₙ

watch c until guard end
```

The `watch` statement, evaluated the condition **c** (created by using `condition()` and signaled by `signal(c)`), and, if **guard** evaluate to true, terminates leaving the lock.

Example

The following example, taken from [Car94] illustrate a pocket calculator, and how to use it.

```
let calc =
  {
    arg => 0.0, (the visible argument display)
    arc => 0.0, (the hidden accumulator)

    enter =>    (entering a new argument)
      meth(s, n)
        s.arg := n;
        s
      end,

    add =>      (the addition button)
      meth(s)
        s.acc := s.equals;
        s.equals := meth(s) s.acc+s.arg end;
        s
      end

    sub =>      (the subtraction button)
      meth(s)
        s.acc := s.equals;
        s.equals := meth(s) s.acc-s.arg end;
        s
      end

    equals =>   (the result button)
      meth(s) s.arg end,

    reset =>    (the reset button)
      meth(s)
        s.arg := 0.0;
        s.acc := 0.0;
        s.equals := meth(s) s.arg end;
        s
      end };
```

An example of the usage of the calculator is as follows.

```
calc.reset.enter(3.5).equals;
(3.5)
calc.reset.enter(3.5).sub.enter(2.0).equals;
(1.5)
calc.reset.enter(3.5).add.equals;
(7.0)
calc.reset.enter(3.5).add.add.equals;
(10.5)
```

3.3.3 Object Migration

The migration of objects is one of the good features of the language. But it is not made directly, and it is supported as weak mobility. Weak mobility means in this case that an object can move to other site only if it is idle, *i.e.*, not while it is executing an operation. Strong mobility mean that objects can migrate while they are busy in an execution, continuing the execution in the new placement. To achieve migration in Obliq, we do it in two phases: (1) by causing the engine to remote clone the object, and (2) by aliasing the original object to its remote clone. In the following code the methodology is illustrated.

```
let migrateProc =
  proc(obj, engineName)
    let engine =
      net_importEngine(enginName, Namer);
    let remoteObj =
      engine(proc(arg) clone(obj) end);    (1)
    redirect obj to remoteObj end;         (2)
    remoteObj;
  end;
```

As it is critically that the two phases of the migration be executed automatically, it is recommended to serialize the object, invoking the `migrateProc` as it is done in the following code.

```
let obj1 =
  { serialized, protected,
    . . . (some fields)
    migrate =>
      meth(self, engineName)
        migrateProc(self, engineName);
      end };

let remoteObj1 = obj1.migrate("Engine@site1")
```

We saw that in Obliq we can "migrate" an object by cloning and then updating its reference. In the following section we review Borg, a distributed language that works using agents for mobile computing, and provides weak and strong mobility, offering to us the best option to create our conceptual language by extending it.

3.4 Borg: Agents and Mobility

Borg is a mobile multi agent platform built on top of Pico [D'H, DM00], a functional language developed by Prof. Theo D'Hondt at the Programming Technology Lab of Vrije Universiteit Brussel. Borg is mainly developed for Linux, but can be easily portable to other operative systems. Several versions for different platform can be found in its web site: *http://borg.rave.org*

3.4.1 Borg architecture

Borg consists on two parts described bellow, the Borg core and the user interface.

- **The Borg core**: It mainly consists in three parts: the virtual machine, the router, and the natives sockets. The virtual machine translates Borg code to machine code to execute the agents. The router uses TCP/IP functionality, allowing communication between two different Borg platforms. The router uses a name server to keeps all agent current locations, ensuring that two agent will communicate using the shortest path. The native sockets allow to communicate Borg platforms with other non Borg platforms, using TCP/IP sockets, allowing communication with servers of mail, web, ftp, etc.
- **The Borg UI**: The user interface of Borg is not portable, and it is specific platform implement. For pur research we used the UI developed for Linux using KDE1. The UI provides an environment were new agent can be created, and agent interfaces to communicate with Borg agents.

The current Borg architecture features are described bellow.

- **Strong Mobility**: Borg agents are able to migrate to another address space in the middle of its execution. This is thank to the ability of Borg to reify the complete computational state of a running process, including its runtime stack. Some other multi agent system can only migrate objects when they are idle, what is call weak mobility.
- **Communication**: Agents always communicate asynchronously, unless the programmer force synchronization. It is important that agents are designed autonomous, then, message sending shall be asynchronous. In other case it would transfer the control flow to another agent.

- **Hierarchical naming system**: Every agent has a human readable name to be referenced. Then, we can do late binding between agent at runtime.
- **Garbage Collection**: Borg incorporate in its system this feature based on the Pico's garbage collector, which is highly performed.
- **Synchronization of agents**: Agents are synchronized using rendez-vous. In the following subsection we describe how to synchronize agents.

3.4.2 Programming in Borg

Basics

Borg is dynamically typed, and using the same syntax of Pico we can use numbers, text, tables, functions and references to agents. A variable containing a number can change to a text variable, or to a table during execution. The following code describes definition and assignment of variables.

(definition)
`a:7`
:7
(assignment)
`a:="hello"`
:hello
(definition of a table)
`t[5]:0`
:[0 0 0 0 0]
(assignment of a cell in the table)
`t[4]:=7`
:[0 0 0 7 0]

To declare a function it has to be done as follows.

(definition of the function)
`sum(a, b)::a+b` *:<function sum>*
(invocation of the function)
`sum(3, 4)` *:7*

The references to the object will be review bellow in the naming system.

Objects

Borg does not use classes to create objects; it is a prototype-based language in that sense. Objects created in the same agent share the address space, and they are not able to move independently. They belong to the agent where they are created. An example of a point object in Borg is illustrated in the following code. Note that it is possible to use mixins like in Agora. In the example we can extend the point to a circle.

```
point(x, y)::{
  getx()::x;
  gety()::y;
  setx(nux)::x:=nux;
  sety(nuy)::y:=nuy;
  show()::display("x: ",x," y:",y);

  circle(r)::{
    getr()::r;
    setr(nur)::r:=nur;
    clone();
  }
  clone()
}
```

Note the usage of the `clone()` operator at the end of the definition of point, and circle. It means that when we invoke point, clone will be the value returned. To create a point, we do it by executing `p:point(1,1)`. Then, to get a circle extended from point `p`, we do it by invoking `c:p.circle(7)`.

Naming

Most of the current mobile systems use an internet name to refer an agent, but if the agent move to another location, then, the name of the agent also change. This is highly undesirable in mobile computing. We want to have a name to refer an agent, and if the agent migrate we still want to refer it with the same name, having transparency, and allowing the agent to move without informing us. Thus, naming system in Borg follow these two rules.

1. An agent has a name, which is unique.
2. The name does not change wherever the agent migrate.

To get a reference to an agent named *machine/agent_pong*, we use the primitive `agent` as follows

```
a:agent("machine/agent_pong")
:<reference machine/agent_pong>
```

Creating and moving agents

There are two ways to create agents in Borg. One is using the graphic user interface clicking on the *new* button. This action will create two agents. The computing agent

(standard Borg agent) and the user interface agent window[2] connected to the computing agent. The other way is by using the Borg primitives. We are going to review *agentclone* and *clone2agent*.

- *agentclone(ag, where)* make a complete copy of the environment of the agent `ag`, and put it in a new agent named `where`. As the name of the agents are unique, if an agent named `where` already exist, then *agentclone* returns void.
- *clone2agent(where)* make a complete copy of the current agent environment, and put it a new agent named `where`. If an agent named `where` already exist, then a random number is assigned as a concatenation of `where` creating an agent with that name. The *clone2agent* operator will iterate assigning random numbers until find a named that is not in use to ensure the success of the cloning.

In the following example, we get the references of two existing agent named `machine/ag1` and `machine/ag2`. We clone the first one in a third one name `machine/ag3`. Then, we clone the current agent to a new clone name `machine/ag2`, but as this not possible, we get a new name.

```
a:agent("machine/ag1")
:<reference machine/ag1>
b:agent("machine/ag2")
:<reference machine/ag2>
c:agentclone(a, "machine/ag3")
:<reference machine/ag3>
d:clone2agent("machine/ag1")
:<reference machine/ag1-777>
```

To migrate an object to a different address space, Borg provides the *agentmove* operator. In the following code, we get a reference of an existed agent named `machine/ag1`, and we migrate the current object to the address space of that agent.

```
a:agent("machine/ag1")
:<reference machine/ag1>
agentmove("machine/ag1")
```

Synchronization

To synchronize two or more agents, Borg provides a primitive with the syntax *sync(ag, tag)*, synchronizing the agent `ag`, with the tag `tag`. This will stop the execution of the current agent that invokes this primitive, until the agent `ag` synchronize with it. The following example shows how synchronization works.

[2]Screenshots of the window can be found in the Chat implementation using P-Borg in section 5.2.3

Agent1	Agent2
`a2:agent("Agent2")`	`a2:agent("Agent2")`
`sync(a2, "tag")`	`...`
`...`	`sync(a2, "tag")`
continues	continues

Message Sending

The main instruction to send a message to an agent is the arrow `->`. It sends the messages asynchronously and automatically returns *void*, then, the current agent can continue its execution without waiting for the reception of the message in the other agent. In this way we can deal with problems like latency and partial failure without stopping the execution of the agent that sends the message. The arguments of the `->` operator are evaluated in the environment of the sender of the message. We use it as follows.

```
a:agent("machine/agent_pong")
:<reference machine/agent_pong>
a->display("Hello Mr.  Pong")
:<void>
```

As we always get *void* with this instruction, if we need a result of the message, we need to use the *callback* technique. In the following code we define a callback to the sender in the agent named `machine/agent_pong`.

```
callback(ag, msg)::ag->display(msg)
:<function callback>
```

Then, we execute in our current agent the following code.

```
a:agent("machine/agent_pong")
:<reference machine/agent_pong>
a->callback("Hello Mr.  Pong")
:<void>
Hello Mr.  Pong
```

Borg also provides other primitives to send and receive messages: *send(ag, msg)* send asynchronously the message `msg` to the agent `ag`, returning the message immediately and the current agent can continue its execution; *recv(ag, pattern)* receive a message from the agent `ag` with a particular `pattern`. It returns the message sent by `ag`. But if no message was sent, it returns *no message*. We can use the wildcard *any* as it is shown in the following example.

Agent1	Agent2
`a2:agent("Agent2")`	`...`
`send(a2, "hello")`	`...`
:hello	`...`
continues	`recv(any, any)`
	:hello
	`recv(any, any)`
	:<nomesg>

Borg also provides synchronized message sending with the syntax: *send(ag, msg)* and *recv(ag, pattern)*. The semantics is equivalent to *send* and *recv*, but this primitives will stop the execution of the agent as it was described with the *sync* operator. Indeed, the implementation of these two primitives is as follows.

```
ssend(to, msg)::{          srecv(from, pattern) ::{
send(to, msg);             sync(from, pattern);
sync(to, msg)              recv(from, pattern)
}                          }
```

In this section we have reviewed the Borg language presenting its principal features, and how program with it. We can see that Borg deal very good with agents, but they do not have another relationship between them than a reference. Thus, big programs become very hard to maintain, having agents moving through the network and referencing each other in an unstructured way.

In the next chapter we are going to extend the language to get P-Borg, a conceptual language to apply the prototype-based programming techniques, to reason about how prototype-base concepts can help us to organize agents in a mobile multi agent system.

3.5 Summary

In this chapter we have reviewed the background in distributed and mobile computing that we are going to use to produce our conceptual language. First, we presented the main concepts that call us to take distribution seriously. We used that concepts in section 3.2.2, to argue about why we do not use an approach like CORBA and others that have the vision of unifying distributed objects. We saw in section 3.2.1 how complicated is the structure of Java RMI to handle type checking in a distributed environment, and why we do not use classes for our purposes.

We consider Obliq a good object-oriented language for distributed computing, and we present it in section 3.3. As we are going to extend Borg, a very good language to program mobile systems using agents, in section refborg we reviewed its architecture, features, and how to program with it.

In the next chapter we are going to present P-Borg, our conceptual language that extends Borg, adding prototypes in a way that can be applied to organize agents, providing a hierarchy of objects with extensions that can migrate through the network.

Chapter 4

Applying Distributed Delegation

In this chapter we present P-Borg, our conceptual language to apply prototype-based programming techniques, in order to organize mobile systems. We extended Borg using the features provided by the language and by Pico, the base language of Borg. The methodology of the extension is explained in section 4.1.

In section 4.2 we present our first analysis of the problem, having important consequences that finally guide us to the definitive model of P-Borg, and that generates interesting future work.

P-Borg is described in detail in section 4.3, given the syntax of the grammar and the main futures of the language. Apart from the *self* variable, we introduce the pseudo-variables *us* and *yield*, that will help us to give more features to our hierarchical structure of objects.

4.1 Extending a language

As we are going to extend Borg, let us review some concepts about how to extend the language. There are some decisions that need to be taken when we want to extend a language, such as the kind of functions that will be used. In this section we present a brief analysis comparing procedures and macro functions, choosing the one that helps us better in the implementation of our system. Then, we present how Pico provides these functions, and how its structure allows us to do meta programming.

4.1.1 Macros and Procedures

When we want to extend a language, we can do it by using macros or procedures[1]. It is important to identify the distinction between both, and, depending on why we want to extend the language, make the right decision. There are two main differences between macros and procedures: evaluation order of the arguments, and scoping.

[1]We are not making a distinction between procedures and functions, as some languages such as Pascal do. For us they are the same.

Evaluation order

In a procedure, the arguments are evaluated when the procedure is called. In a macro, the arguments are not evaluated, and the programmer decides when the evaluation will happen. That feature of macros allows us among other things, to return a function and then, when the new function is called, evaluate the arguments.

Let us take the example of an *if(condition, then, else)* function to prevent a division by zero.

```
if ( z=0, error, 1/z)
```

If the function *if* was a procedure, all the arguments will be evaluated when the *if* is called. Then, the third argument: **1/z**, will also be evaluated generating an error in case of **z** equal to **0**. In this case it is necessary to use a macro to implement *if*, because in that way we can decide when to evaluate the *then* or *else* argument, depending on the *condition* argument. Thus, just in case of **z** not equal to **0**, we evaluated **1** divided by **z**, preventing the division by zero.

Scoping

Calling a macro, the arguments are evaluated in the context of usage of that macro. For instance in Scheme, the expressions that *if* received as arguments, are evaluated in the context where the *if* appears.

Procedures are lexically scoped, i.e., calling a procedure, the arguments are evaluated in the context where the procedure is defined. In the *fibonacci* function, every call to the function will evaluate the arguments in the context where *fibonacci* is defined. Thus, the **n** value of the recursive calls to *fibonacci*, will correspond to the value received as a parameter of the function.

```
fibonacci(n) :
        if ( n>1,
              fibonacci(n-1) + fibonacci(n-2),
              1)
```

Summarizing, procedures evaluate their arguments when the procedure is called. Macros do not evaluate the arguments. Procedures make the evaluation in the context where the procedure is defined. Macros evaluate in the context where the macro is used. We are going to use mainly macros to develop our system, due to the fact that we need to build methods which arguments will be only evaluated when they are sent to the objects as messages. For instance, *method* will be a macro that will receive the body of the method as an argument without evaluating it, but generating the method as a result of the calling. When we send a message to an object for applying the method, then the body of the method will be evaluated while executing the method.

The other reason to use macros is to implement delegation. We need to bind the *self* variable of the object to the original receiver of the message. Thus, when the message is performed and the body of the method is evaluated, we need to evaluate it in the context of the receiver of the message, even when the message is delegated to another object. This will be clearer in the following sections of the chapter.

4.1.2 Macros in Pico

Pico[2][D'H, DM00] is a very small functional language and also very powerful. Borg is an agent language developed on top of Pico. As we want to extend Borg, we will present in this section how macros work in Pico, and how we can do meta-programming in it.

In Pico, there are no special keywords to define procedures or macros. The evaluation of the arguments depends on how we define their reception. A function defined as **fun(**arg_1**,** arg_2**, ...)** behaves as a procedure, and the arguments will be evaluated when the function is called. Meanwhile, a function defined as **fun(**arg_1**( ),** arg_2**( ), ...)** behaves as a macro, where the arguments will not be evaluated. We can even have functions like **fun(**arg_1**,** arg_2**( ))**, where the arg_1 will be evaluated when the function is called, but arg_2 will not.

Let us look at the example of the boolean functions. The following two expressions correspond to a definition of the method. Both are macros, then, they do not evaluated the arguments when they receive them. True function ignores the second argument, and returns by calling the first argument. False function returns the second argument, calling it and ignoring the first argument.

```
true (x(), y() ) : x()
false(x(), y() ) : y()
```

This two boolean functions allow us to create the logic functions *and*, *or* and *not*. The *and* function evaluates the preposition p. If p is true, then, evaluates the preposition q, returning its value. If the value of p is false, we do not need to evaluate q, then, we return false. The *or* function works similar. If the proposition p is true, we immediately return true without the necessity of evaluating q. We just evaluate q if preposition p is false. The *not* function evaluates p, and then, if p is true return false, and vice versa.

```
and(p, q() ) : p( q (), false)
or(p, q() ) : p( true, q())
not(p) : p(false, true)
```

A very interesting example is the formal definition in Pico of the function *while*, that receives two arguments: a condition and a expression. This macro function has only

[2] Pico stands for 10^-12 and hence very small; the name has no other meaning.

two expressions. The first one is the definition of a new procedure named **loop**, and the second one is the first calling of the **loop** method with void as a value, and the condition as a predicate.

```
while(cond(), express()) : {
   loop(value, pred) :
        pred( loop(express(), cond()), value);
   loop(void, cond()) }
```

As we can see in the code, loop receives a value and a predicate as arguments. If the predicate is true, the loop continues because the procedure is called again with the evaluation of the expression as a value, and the condition as a predicate. The loop stops when the predicate is false and the value is returned. The code presents a procedure (loop) into a macro (while). The condition and the expression are just evaluated when becoming the parameters of the loop procedure. Notice again that the difference between macro and procedure is just terminological. Technically they are just Pico functions.

4.1.3 Using Meta Programming

We are going to use the macro function of Pico, but we will also need to manipulate special variables to organize the objects, and to implement the distributed delegation. Then, we will need to do meta programming.

Meta programming is a technique that allows us to reason about and manipulate another program. To be able to do this, we need to have access to the construction level of the language, and modify the program when we need it. With the structure of Pico we can have this access. Now we will explain what we are going to need from Pico to do meta programming to implement our system.

In Pico there are values, variables, functions, tables, dictionaries, etc. For the purpose of our thesis, we need to access mainly the dictionaries and functions. The dictionaries will represent our objects. To implement delegation between objects, we will set our special variables in the functions. We must consider that there are different versions of Pico and Borg. The explanation goes according the version we used for the experiment of the thesis.

Functions are represented with a table of three elements. The first element has the reference to the function. The second element is a table with the arguments of the function. The body of the function can be accessed in the third element of the table. We will concentrate on the body of the functions, because we are going to install our variables there. More details about our way of proceeding will be given in the following sections with the correspond reasoning. The following example will help us to understand how functions are structured on tables.

```
fun(x, y) : {
      z : x+y;
```

```
        display(z)
    }
```

The code corresponds to the definition of the function **fun**, receiving two arguments, **x** and **y**, that will be stored in a table. That table will be in the second element of the table that stores the function. The body of **fun** is a begin application (with the curly brackets). The begin application is built with two elements. The first one is the reference to *begin*, and the second one is the body of the application.

In table 4.1, we can see the three elements of the table of the function **fun**. The second column corresponds to the evaluation of the elements of the first one, and the third column corresponds to the meaning of the element. Remember that this representation corresponds to the version of Borg that we use for the experiments, and it is possible to find other versions of Pico with a different representation.

Element	Evaluation	Meaning
`fun`	<*function fun*>	The function fun
`fun[1]`	<*reference fun*>	Reference to the function
`fun[2]`	<*table*>	Table of arguments
`fun[3]`	<*application begin*>	Body of the function

Table 4.1: Elements of the table of a function in Borg

As we said in the previous paragraph, an object is a dictionary in Pico, therefore in Borg. The table in Borg will be very similar to the one of the function, but it will take four elements. The fourth one will be the environment where the object is created. In Borg, the environment will be the agent where the object is placed. The first three elements are the references to the dictionary name, the initial arguments of the object, and the body.

The following example presents the definition of a counter object, and then **counter** will be a counter object. Note the cloning function at the end of the definition of the object. It is like it was explained in section 3.4.

```
{
  make_counter():: {
     c : 0;
     increment() :: c:=c+1;
     getC() :: c;
     clone()
  };

  counter:make_counter()
  }
```

As in the function, table 4.2 details the structure of the object table on Borg. Note that the second element is void, because the counter does not receive arguments to be created.

Element	Evaluation	Meaning
`counter`	*<dictionary>*	Counter is a dictionary
`counter[1]`	*<reference [dctname]>*	Reference to the dictionary name
`counter[2]`	*<void>*	Table of arguments
`counter[3]`	*<dictionary>*	Body of the object
`counter[4]`	*<dictionary machine/Agent >*	Environment of the object

Table 4.2: Elements of the table of an object in Borg

In section 2.2, we explained that in delegation, the whole dictionary of the original receiver of the message is passed to the parent object. This is done because we need to bind the *self* variable to the receiver of the message, and not to the object on which we are delegating, executing the message in the context of the receiver. As in Borg we can find the body of the object in the third element of the table of its representation, we use it to pass the dictionary of the object to the parent object.

In the following section we present the first model of our system, explaining how and why we evolve to the definitive system presented in section 4.3.

4.2 First Analysis

In the previous section of this chapter, we did a review of macro functions in Pico, and some features at the construction level of Borg, to do meta programming on it. Together with the prototype-based programming theory, we introduced the Split Objects and Agora with the mixin methods. The background in distributed system, and languages such as Borg and Obliq, were presented in chapter 3. Now, we will present our extension of Borg, to apply the prototype-based techniques to organize the mobile system.

We consider the approach of Split Object a good way to organize the delegation of behavior and properties, creating hierarchies as the primary structure of the system. The delegation is done inside the object, but not between different objects. The delegation inside the object means that it is done between the extension and the “root” object, as it is done in Agora between the mixins (views) and the base object. But in Split Object, we can not access directly the extensions of the object (called pieces). We can only do it from inside the object. This is because split objects are considered the single entity of the real world, and then, first class entities. In our approach, we want to access every extension of the object, because we consider them as individual objects, as in Agora. Then, what we are going to implement in our extension of Borg, is delegation with distributed mixin methods.

4.2.1 Delegation with Local and Distributed Mixin Methods

We already have saw in section 3.4, how we can create objects and mixin methods in Borg using the *clone* operator. Now, we want to implement delegation between the extension of the object, and implement a distributed mixin method. To implement delegation, we need to add a *super* variable to generate a "parent-link" relationship, and implement the method-lookup mechanism. To deal with distributed mixins, we will use the *clone2agent* operator, which will make a clone of the dictionary placing it at a new agent. To explain the reasoning and the implementation, we will use the example of a point that will be extended to a circle.

To create an object in Borg, we have seen that we define it as a normal method, but at the end of it, we add the *clone* operator as the value of the expression. Then, a copy of the scope of the method will be returned as a dictionary. Now, we must install a *super* variable to generate the relationship with the parent object. The *super* in the object generated by invoking the mixin method will refer to the main object, and if the mixin is inside another mixin, the *super* will refer to the parent object created with the mixin that contains the other object.

We need to bind the *super* variable dynamically to the receiver of the message, every time the mixin method is called. Then, we need to capture the execution of the method, set the *super* variable, and then continue the execution.

Let us suppose we have a point *p*, and we decide to extend it to the circle *c*. When the message *circle* is sent to *p*, we bind the *super* variable of *c* to *p* before perform the message, and then we generate the extension. As *p* is the receiver of the message, while we are executing the *circle* method, the *self* variable corresponds to *p*. Then, we set *super* as *self*, and then we continue with the execution of the method.

To create a distributed extension, we use the same mechanism, with the only difference that instead of using the *clone* operator, we use clone2agent, placing the copy of the dictionary in a new agent as it was explained in section 3.4. Nevertheless, this small difference will produce important consequences, taking us into a discussion at the end of this section, which finally will lead us to the definitive model of our system. The first consequence lay on the difference of the result of the two cloning operators. Meanwhile *clone* returns a dictionary, *clone2agent* returns a reference to the new agent created. This difference is more that only a technical consequence, because it means that local extensions share the address space of the parent object, while a distributed extension has its own address space, as it is desired in agents.

In the introduction to Borg, we already reviewed the two different operators to send messages to a dictionary or to an agent, "." and "->" respectively. We will avoid this difference introducing *sendo*[3], an operator that allows us to send messages to agents or dictionaries (objects).

The *sendo* operator will be useful for two purposes. First of all, as we already mentioned, it will avoid the difference of sending a message to an object or to an agent. As

[3] The name *sendo* means "send to object", and we wanted to differentiate it from *send*, a Borg's primitive explained in section 3.4

our primary aim of this work is to provide a way of organizing mobile system, we will call “objects” to all of our entities (This is not unifying local with distributed objects, we will analyze the difference). Second, it will allow us to capture the message sending to manipulate it. We want to use our own method-lookup, and install the hidden variables to create the parent-link relationship to implement delegation.

We provide two other operators to send messages: *sendoBack* and *sSendoBack*. We will explain them in detail in section 4.3.4. In this section, we will only use *sendo* to explain the model. Its syntax is the following, where **object** is the receiver of the call, **method** is the name of the method that will be executed, and the third argument corresponds to the arguments of the method.

```
sendo(object, method, [arg1, arg2, ... ])
```

Now that we are able to capture the message sending, the delegation works like follows:

1. Check if the message is intended for a dictionary or an agent, and then continue the sending. In the case of the agent, we first need to prepare the message before sending it, but this will be explained deeper in the definitive model.

2. Once we are in the dictionary of the object, we start to look for the method. At this point it is important to have the notion of two variables: *self*, that will always be the original receiver of the message, and *me*, corresponding to the actual dictionary where we are looking for the method. At the first searching of the method, these variables correspond to the same object.

3. If we find the method, we apply it in the context of the original receiver, *i.e.*, with the dictionary of *self*.

4. If we do not find the method, we look for the *super*. If there is no super, we generate the error message: “Message not understood”. If we find the *super*, we delegate the message to it, passing the whole dictionary of self, to execute the method in the context of the original receiver. Thus, *self* does not change along the method-lookup, but *me* becomes the super variable every time we delegate the message.

Method	**Description**
`obj("name", body() )`	Definition of an object
`method( body() )`	Method, receiving the body as a parameter
`mixin("name", body() )`	Local Mixin-Method
`netmixin("name", body() )`	Network Mixin-Method

Table 4.3: Methods of the extension of Borg

Considering we are going to manage the applying of the methods, we provide a new way to define them in the code. Within the *method* we can use normal Borg code. As mixins and network mixins are also methods, but with special features, we also provide the way to define them, eliminating from the programmer the setting of the *super* variable and the corresponding cloning. We are going to do the same to create objects. The syntax of the set of methods that we provide in our extension is presented in table 4.3.

4.2.2 Sharing or not address space

More than just a technical consequence the different between *clone* and *clone2agent*, we considering the question of sharing address space an important point of analysis. Let us consider the following code of a point that can be extended to a circle using local and distributed mixins.

```
point(x, y) :: obj("Point", {
    setx(nx) :: method( x:=nx );
    sety(ny) :: method( y:=ny );
    show() :: method(display("x : ", x, " - y : ", y));

    circle(r) :: mixin("circle", {
        setr(nr) :: method( r:=nr);
        `first, send the show message to super
        `then show r
        show() :: method({sendo(super, show, []);
                    display(" - r : ", r)})
    });

    netcircle(r) :: netmixin("netcircle", {
        setr(nr) :: method( r:=nr);
        `first, send the show message to super
        `then show r
        show() :: method({sendo(super, show, []);
                    display(" - r : ", r)})
    })
})
```

Consider we have a point *p*, and then we extended to a circle *c*, and to a netcircle *nc*. The result of this execution is depicted in figure 4.1. Only the cloning is occurring in a distributed way, but not the delegation. This occurs due to the fact that we are setting a dictionary as the super of an extension, whether is local or distributed, and the *clone2agent* operator clones the completely dictionary to a new agent.

Local objects shared the same address spaces, and this is what is occurring between **p** and **c**. But in this way we are not allow to migrate the objects independently. They

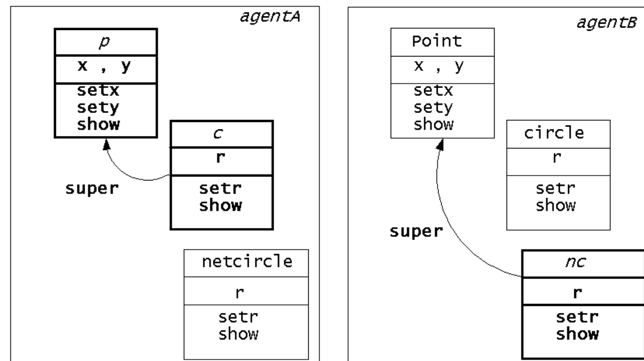

Figure 4.1: The difference between *clone* and *clone2agent*.

are attached to the agent, and even when is not the same situation than the dependency between and class and its instances, we fall in a similar problem on having non-mobile self-supported objects. If we want to apply prototypes to mobile systems, we must make them behaves as agents.

Even though, we do not consider this first approach useless, because it also provides a possible advantage of the deep copy of the dictionary in the new agent, that can help us to delegate some messages locally, reducing network traffic. It can also be useful in systems when we want to have a new object independent of the original one, having its own state.

But the most useful application that can be derived from having local and distributed extensions, it is a sort of "distributed split objects", where the local extensions become the pieces of the agent-object, being only accessible within the address space, but not directly accessible from other agents. The idea is depicted in figure 4.2

4.3 P-Borg

One of the features of Obliq, is that every object is potentially and transparently a network object. In our system, the local extensions can not be considered as network objects, and not even their parent object. They are not referenced from other address spaces, and they just move if the agent where they reside move. To send messages to objects in Obliq, we use *network references*. In Borg, only agents are known by *references*, but not the objects. Thus, to solve the problem presented in the previous section, we are going to encapsulate every one of our objects in a different agent. Then, having a reference to an agent, we are having a reference to an object.

4.3.1 Only Network Mixin Method

Considering the previous argument, we take the decision of encapsulate the objects into independent agents, having only one object per-address space. Technically, in our imple-

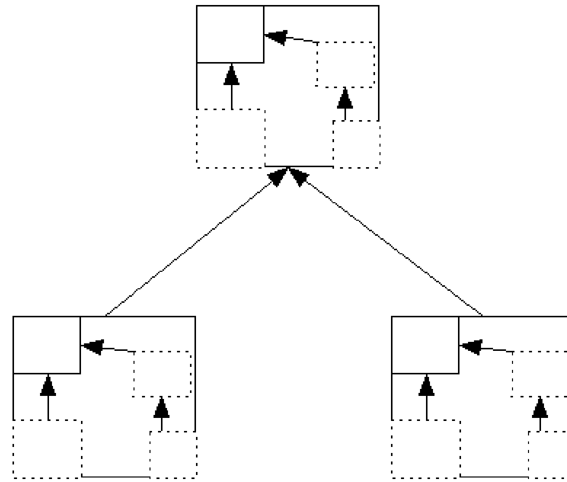

Figure 4.2: Distributed Split Objects.

mentation we are not going to use the *clone* operator anymore, only *clone2agent*. Even to create a new object we will generate a new agent where the object will reside. Every object will be a distributed object known by the reference of the agent where the object lives, and every extension will be a new object living in a new agent.

As another consequence of this new design, every time we create an extension, we will consider the *agentself* variable to set the *super*. Even though, the conceptual value of *self* in the delegation mechanism does not change. We are considering *agentself* only with the purpose of setting an agent as the *super* variable of an extension, instead of set a dictionary as in the first model. The new system is depicted in figure 4.3.

One of the important advantages of this new model, now that every object resides at its own agent, is that we can use all the features that Borg provides for agents in our objects. We can move them through the network in a transparent way using the *agentmove*, and the object will always be known by the reference generated with the *clone2agent* operator.

The mobility of the agents, whether strong or weak, it is not the merit of this work. But, we provide a way to organize the agent as prototype objects, delegating behavior and properties between them, and having a hierarchy derived from the usage of the extensions. We provide a structure for mobile code that is better maintain, understand and evolve.

4.3.2 Implementing Distributed Delegation

How does the new design of the model modify our delegation system? Principally, it modify the algorithm in two points, but not critically. First, we said that we capture the message sending, and then we check if the receiver is a dictionary or an agent. Now this checking is not longer necessary. We just apply the way of sending a message to an agent. Second, every time we must delegate the message to the parent object, we must do

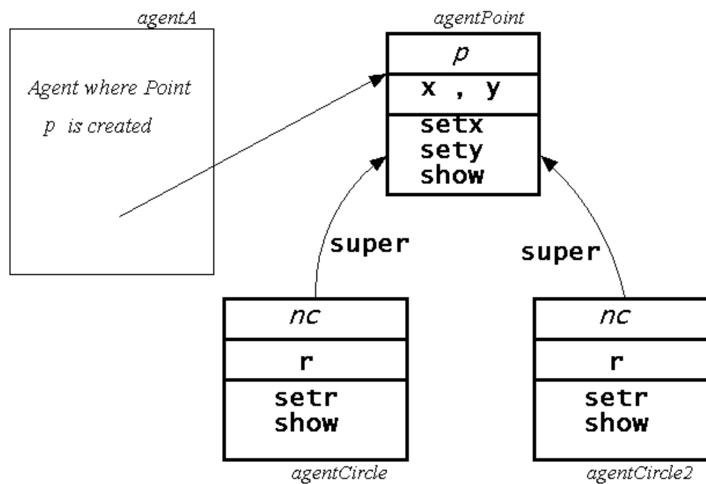

Figure 4.3: The definitive model.

it as sending a message to an agent, but do not forgetting to send the environment of the original receiver of the method.

We already have seen the mechanism of sending a message to an agent in the introduction to Borg in section 3.4. In our system, if we send the message *m* to an object (that now is an agent), the only difference is that we are always sending the message *delegate*, and giving the name *m* of the method as a parameter, together with other internal variables of the system. We are doing this by filtering the sending though the our function *delgate2agent*, that it is executed in the environment of the sender of the message. The function *delegate* performs the delegation method-lookup in the receiver of the message. If *m* is not found, we delegate it to the *super* through *delegate2agent*, passing the self environment. If *m* is found, we apply the method with the arguments given by the sender of the message.

If we come back to the code of the point extensible to a circle, we can see that for the programmer it looks the same as the first model, but now we do not have local mixins. Now, we also add the methods *down* and *yesDown*, to show that the execution of the method is made in the context of the receiver of the message

```
point(x, y) :: obj("Point", {
    setx(nx) :: method( x:=nx );
    sety(ny) :: method( y:=ny );
    show() :: method(display("x : ", x, " - y : ", y));
    down() :: method(sendo(self, yesDown, []));
```

```
    netcircle(r) :: netmixin("netcircle", {
        setr(nr) :: method( r:=nr);
        show() :: method({sendo(super, show, []);
                        display(" - r : ", r)});
    yesDown() :: method(display(super))
    })
})
```

Now we create a point *p*, and we extend it to a netcircle *nc*. In the following code we use *sSendoBack* to get the extension. That operator will be explained in detail section 4.3.4. Now, just considered as a way to send a message to an object, receiving the result back. Both are reference to the new agents where the objects reside. Then, we send two messages to the *nc*.

```
p : point (1, 1)
nc : sSendoBack(p, netcircle, [5])
sendo( nc, setr, [ 3] )
sendo( nc, setx, [ 0] )
```

The first message will be performed by *nc* without delegating to *p*, meanwhile the second one will be delegated. The two situations are depicted in figure 4.4 and figure 4.5.

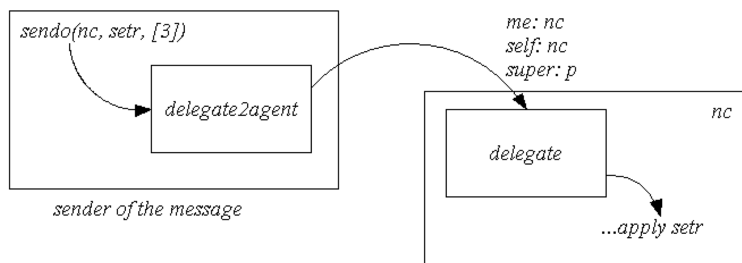

Figure 4.4: Sending "setr" to a netcircle.

Now if we send the message *down* to *nc*, the method will not be found in the object. Then, it will be delegated to *p*. The point will find the method and will execute it in the context of the receiver, *i.e.*, in the context of *nc*. Now the method *down* sends the message *yesDown* to *self*. If the variable *self* is bound to *p*, we will get a "Message Not Understood" error, but as *nc* is delegating the message, *self* is bound to *nc*. Then, the exection of the method send the message *yesDown* to *nc*, finding the method and returning *super*. In the case of the example, returns a reference to *p*. This example show that the environment of the receiver of the message is passed to the parent object every time we delegate, to execute the message in the context of the receiver.

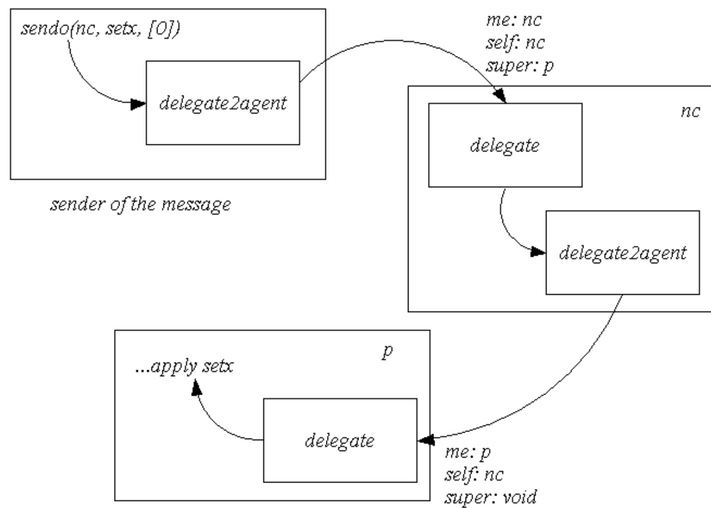

Figure 4.5: Sending "setx" to a netcircle.

4.3.3 The *us* and *yield* pseudo variables

We already introduced the notion of the variables *self* and *super*, and when we explained the method-lookup of the delegation in our system, we identify a conceptual variable me that represents the performer of the method. We can see them in figures 4.4 and 4.5. Now we are going to introduce two new pseudo variables: *us* and *yield*.

- *us* : This variable represents the set of the first level of the extensions of an object. If an object does not have extensions, then the variable refers to itself.
- *yield* : This variable represents the yield of the hierarchy of extensions of an object. If an object does not have extensions, then, it belongs to the yield.

As the extensions of an object are conceptually different objects, we consider these two variables useful for our purposes of organization. The *us* variable is useful to send messages to all the first level extensions from the parent object, it is the network analogue of *self*. Until know, with the delegation system only the extension knows its parent object, but with the *us* variable, the parent object can send messages to its extensions. We are not implementing delegation in the other direction, we just provide a way to send messages explicitly from the parent object to its extensions. Just the first level of the extensions will be included in this variable. To send messages to different levels, we must refer to the *us* variable of the extensions. The variable is depicted in figure 4.6.

The *yield* variable can be very useful, because allow us to access easily and in a structured way, the last objects of the hierarchy. For instance, if we are modeling trees,

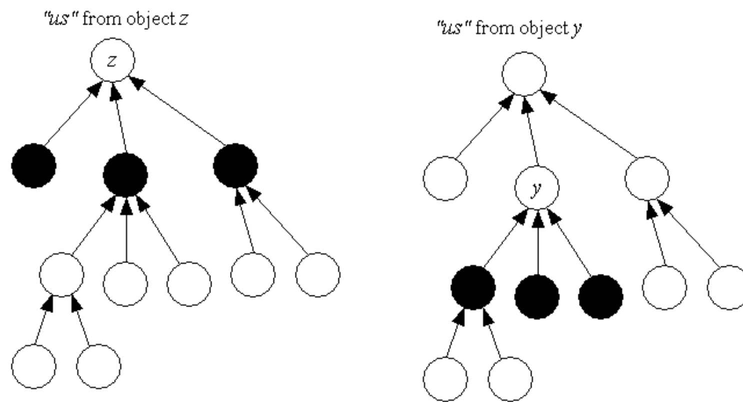

Figure 4.6: The “us” variable.

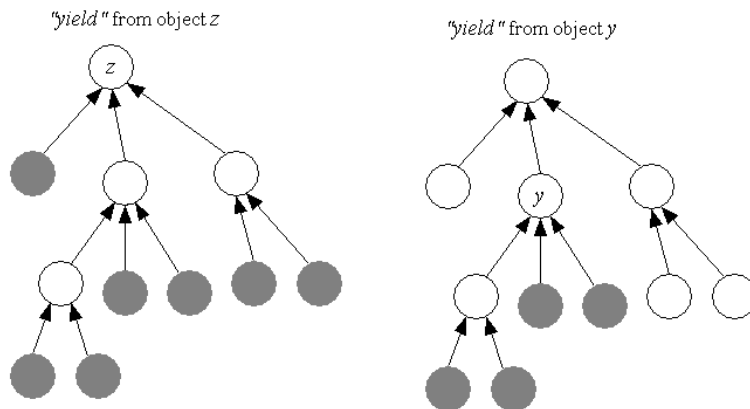

Figure 4.7: The “yield” variable.

with this variable we can send a message to the entire yield just sending a message to the variable. Figure 4.7 presents the semantic of the variable.

To implement these two variables, we do the checking before send the message to the any object. Once we identify if the message is sent to *us* or *yield*, we propagate it to the correspondent objects. Each object becomes the receiver of the message, and the delegation process is performed independently. Note that these two variables are only accessible within the objects. We can not send a message to *us* or *yield* from out of the objects, because they will be out of context.

In the chapter of application of the system, in section 5.3, we present a good example to clarify the advantages of this two variables. In the following section we presents the operator that we implemented to send messages to the objects.

4.3.4 Sending messages to the objects

In the previous sections of this chapter, we already introduce the *sendo* operator to send messages to objects. We also mentioned the other two operators *sendoBack* and *sSendoBack*. In this section we are going to explain them in detail.

The three operators receive the same arguments: the receiver of the sending, the name of the method, and the arguments of the method in the format of a table. Their syntax is like follows.

```
sendo( object, methodName, [arg1, arg2, ... ] )
sendoBack( object, methodName, [arg1, arg2,... ] )
sSendoBack( object, methodName, [arg1, arg2, ... ] )
```

The *sendo* operator is the equivalent of the arrow "->" operator of Borg, it sends the message asynchronously returning always *void*. Asynchronous messages are important in distributed systems considering the latency problem. Several times we do not want to wait for the execution of the method, moreover if we are not interested in capture the result of the execution, for instance, when we set a variable of an object. Do not use this kind of sending if you want to get the return of the execution. It will always return *void*.

The *sendoBack* sends the message asynchronously, but asks the receiver to send the result back, also asynchronously. This operator is useful when we do not want to wait until the end of the execution of the message we sent, but we are interested on receiving the result in a certain moment. Then, we need to explicitly ask for the reception of the message with *recv*, a primitive function of Borg. Imaging we want to send a message to different objects, and we know that every execution will take a long time to finish, but we need to receive the message later. With this operator we can avoid the idle state in the sender of the message, while the receiver is executing.

Even when in distributed systems it is recommended to use asynchronous message sending, sometimes it is necessary to send synchronized messages. Then we use the *sSendoBack* operator. To create extensions of objects, we recommend to use this function instead of *sendoBack*, because we ensure that we will send message to the extension, only after it is created and we have a reference to it.

It is very important to have consistency in the usage of *sendoBack* and *sSendoBack* in the same code. If we use *sendoBack* to send a message to an object *o*, and then we use *sSendoBack* with the same object *o*, without receiving the first message, we can confuse the reception of the two execution, depending on which message of *o* reply first.

Figure 4.8 presents how this three operators are implemented in the case of delegation. In the figure, object *o* send the message to object *c*, that is an extension of *b*, delegating the message to *b*. The object *b* is an extension of *a*, and it also delegates the messages. Finally *a* execute the message in the context of *c*. Note that when you send a synchronized message, all the objects in the chain of delegation stop its execution until the execution is performed.

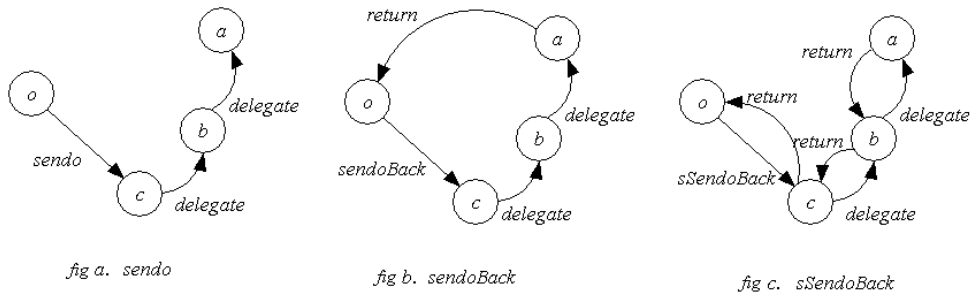

Figure 4.8: The three kind of sending message to an object.

4.3.5 Cloning

In our system, we are just cloning objects implicitly. We do it every time an object or an extension is created, cloning the definition dictionary to a new agent. But we do not offer a way to clone existing objects. The consideration that has to be taken to implement a clone operator has relation with the parent object.

Let us suppose we have an object *b* delegating on an object *a*. If *a* does not have any parent link, clone it is not a problem, we just create a copy of the object in a new agent. The decision comes when we decided to clone *b*. The clone *b'* will keep a parent link to the object *a*, or a new object *a'* will be created as is depicted in figure 4.9.

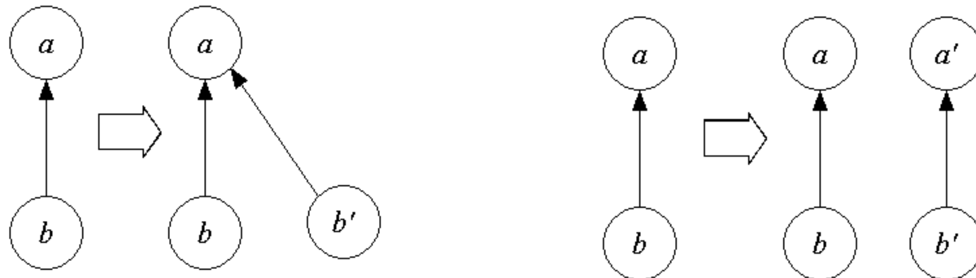

Figure 4.9: The cloning observation.

4.4 Summary

In this chapter we have presented our extension to the language Borg, providing a way to organize mobile agent systems using a hierarchical prototype-based approach, based on objects that can be extending through the network, delegate behavior and properties to the parent object.

First, we introduced the elements that we decided to use to extend the language. We explain the macro function, and the way of doing meta programming in Borg, to know where we must modify the methods to install our variables. More details about the implementation can be seen in the annex.

In section 4.2, we present the first model of the system and the problems that presents, but understanding that can provides some alternatives to improve the definitive model, like implementing local delegation to reduce network traffic, or a new hierarchy of organization near to Split Objects.

The definitive model is explained in detail in section 4.3, presenting the solution to the first model, and going deeper in the explanation of the implementation of the distributed delegation. All the operator of the language are presented, including the sending methods, and the variables *us* and *yield*, offering a new organizational feature to the language. We explain our considerations to implement the cloning of existing objects, and the implications that have in the relationship with the extensions of the objects.

Table 4.4 presents the grammar of the extension of Borg, and a small description.

Variable/Function	**Description**
me	Hidden variable that is only used in the delegation method-lookup, to represent the current dictionary of searching. We can not send message to it
self	Conceptual variable that represent the receiver of a message
super	Variable to set the parent-object
us	Represent the first level in the hierarchy of the extensions of an object. The "first" level depends on the object
yield	Represent the yield of the hierarchy. If an object does not have extension, belongs to the yield
method(body)	Define a method. Receive the body of the method as a parameter
obj(where, body)	Define an object. It will be placed in a new agent named "where". The body of the object it is also a parameter
netmixin(where, body)	Define a network mixin method. Receive the body of the netmixin as a parameter, and the original name of the new agent
sendo(obj, method,[arg1, ...])	Sends a message *method* to the object *obj*, in a synchronous way
sendoBack(obj, method,[arg1, ...])	Sends a message *method* to the object *obj*, in a synchronous way, asking for an eventual result
sSendoBack(obj,method,[arg1, ...])	Sends a synchronized message *method* to the object *obj*, returning the result of the execution

Table 4.4: P-Borg grammar

Chapter 5

Validation

To validate the concepts introduced in chapter 4, we present three case studies in the following section. We tried to choose heterogeneous examples to included different aspect into our study. First, we present an analysis of the Role Object Pattern, to see how distributed and mobile prototypes can help to design general solutions to recurrent problems.

In section 5.2 we present three different implementations for a very well known problem in distributed computing: the Chat Room. We first present a solution using Java RMI, and then implemented in Borg without using our mechanism. The third implementation correspond to P-Borg, showing how agents can be structured it to build a maintainable solution. Finally we used the advantages of mobility to describe how it is possible to reduce network traffic migrating the server agent or the clients.

The third example is the design of a very simple distributed database, showing an alternative and experimental use of our organizational model, enriched with the pseudo variables *us* and *yield*.

5.1 The Role Object Pattern

The applicability of the Role theory [BT79] is presented in many research areas such as distributed system management [KM96], collaboration and coordination, agent[1] and robot systems [Rie98, Ken], and web applications where the system must allow users with different roles to access information, having a different behavior according the role. Let us think about a web page where users have a personalized way of seeing the information. Then, we can model all the personal preferences of the users in roles.

When we have taken the decision of using object-oriented programming for designing role models, the Role Object Pattern [BRSW97] can help enormously to build the system. In this section, we will make a review of the pattern, presenting how to attack the problem from the prototypical point of view, using our system to implement the solution. With this

[1] The agents cited here belong to the Artificial Intelligence area. They are not the same to the agents of our system.

example we want to present how useful our approach can be at the level of design, even more in distributed systems such as web applications.

5.1.1 Intent and motivation

The intent of the pattern is as follows:

> "Adapt an object to different client's needs through transparently attached role objects, each one representing a role the object has played in that client's context. The object manages its role set dynamically. By representing roles as individual objects, different contexts are kept separate and system configuration is simplified".

In chapter 2 we have said that one of the advantages of prototypes is the simplicity, helping us to generate the representation of the problem domain starting from a concrete example. Let us analyze the case study on the motivation of the pattern, which consists on developing software for a banking environment. The problem arises with the necessity of having different clients that need context-specific views on the key abstractions of the model.

The main key abstraction of the model is the concept of *customer*. Thinking about classes, this information easily guides us to have a Customer class. The interface of the Customer class provides messages to manage the name, number of the account and the general information of the customer, and also provides messages to manage operations such as saving or withdrawing money. The problem appears when we want the customer to handle the possibility to behave as borrower or as an investor, or any different role. The solution could be to extend the customer by adding the different roles to the Customer, but if we try to integrate several kinds of behaviors in one class that will be static at runtime, finally, it will be very difficult to maintain and to understand.

The clever solution proposed by the pattern, is to separate the *core object* (our original customer) from the *role objects*, defining an abstract class for customer roles. The abstract class for customer roles has subclasses that will represent the concrete roles that the customer can play. Then, it will be possible to add concrete roles dynamically to the core of the customer. The class diagram is shown in figure 5.1. The object diagram is presented in figure 5.2.

Thinking now in the way of conceiving a system based on prototypes. First we will find a good example of an object in the model. Now the customer appears as the first object (prototype) of our model, and other customers will be created by cloning some of the existent ones. When the necessity of modifying the behavior of a particular customer appears, we will extend it by using a mixin method.

In the previous explanation, we can see that at this point of the design, the class-based approach needs to take a different solution by adding abstract classes to the system with the only aim of adding dynamic features to the model. In our approach, we can deal extending the core object dynamically in a well-organized manner. We can continue

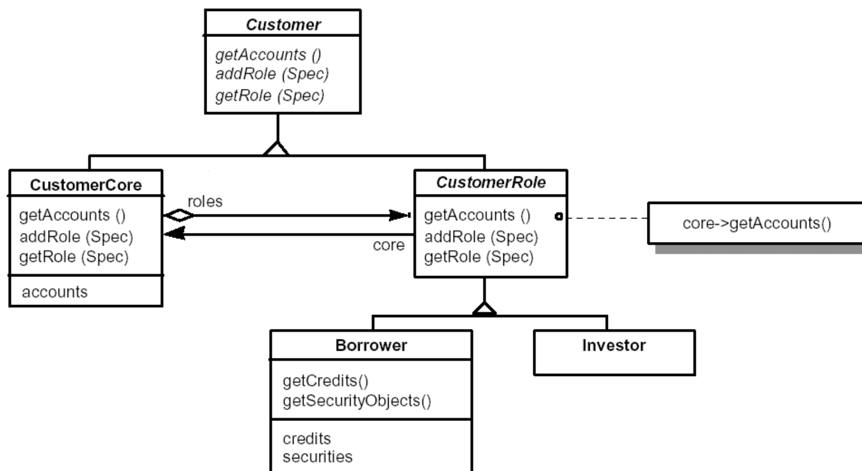

Figure 5.1: Customer hierarchy in a banking environment

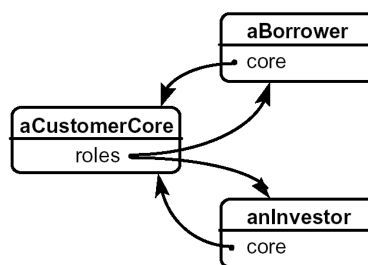

Figure 5.2: An object diagram of the Role Object Pattern

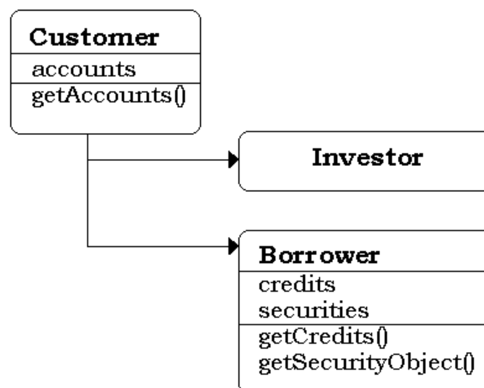

Figure 5.3: The Customer object and its extensions in a banking environment

with the first idea of extending the customer by adding different roles. Each kind of new behavior will be expressed with a new mixin method of the prototype.

The other advantage arises when the message netmixin is sent to a customer at runtime extending its behavior dynamically, and placing the extension in a different address space as an agent. Now we can use the features of the agent system, allowing us to move the role through the net, placing where it is most required reducing network traffic. As the customer is also a mobile object, we can move both wherever we want. This can be very useful if the bank-system that we are modeling will work as a web application, or as a distributed application internal to the bank.

The problem is solved, and even more we are giving extra advantages to implement the model in distributed systems. The model is presented in figure 5.3, showing the customer object and its possible extensions. It looks simpler that the class diagram of figure 5.2, and clearly looks more similar to figure 5.1, which corresponds to the instance objects of the class diagram of the pattern. The similarity occurs due to the fact that prototype models are just conformed by objects.

5.1.2 Structure

To get the general structure of the solution, the following figures show the two possible approaches. The structure of the Role Object Pattern is showed in figure 5.4, and the figure 5.5 presents the structure of the prototype solution.

In the diagrams, the **Component** class represents the customer in the motivation example. The **ComponentCore** is the **CustomerCore**. In the prototype model, we just put the core of the component and the definition of the component object.

The abstract class **ComponentRole** is the generality of abstract class **CustomerRole**, that is not needed in the prototype approach. The **ConcreteRoles** can express the roles of the customer, and in the prototype model they are presented as the extension of the component object.

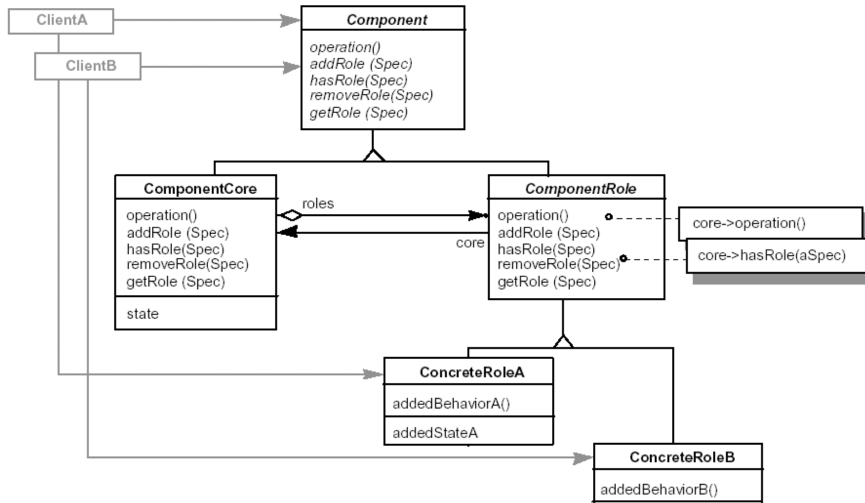

Figure 5.4: Structure diagram of the Role Object Pattern

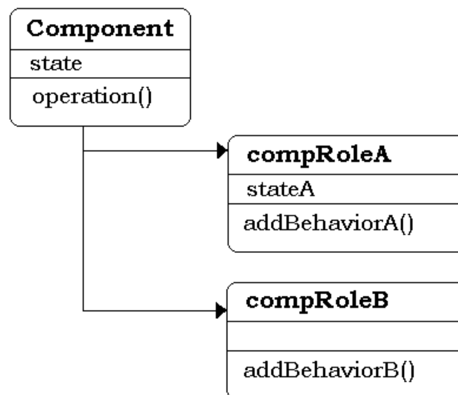

Figure 5.5: Structure of the prototype-based approach

Meanwhile the Role Object Pattern succeeds in defining concisely the key abstraction of the problem domain, in the prototype approach we just need to find the object that represents a good example of the main component of the system. One of the advantages in favor of prototypes is the capability of modifying the behavior of the objects dynamically, but in this case, we find good the way which the pattern is able to change dynamically in a class-based model the behavior of the objects, helping the system to evolve. What we most appreciate in the alternative way of designing using prototypes, is the simplicity of the model, and the advantages that help us to build a better distributed system.

5.1.3 Implementation

The following is a sketch implementation of the structure of the prototype system, using the syntax of our system. The class-based implementation can be found in detail in the role object patter [BRSW97].

```
component( state ) :: obj("ComponentCore",{

    operation( )::method( ... );

    compRoleA( stateA ) :: netmixin("RoleA", {
        BehaviorA( ) :: method( ... )
        });

    compRoleB( ) :: netmixin("RoleB",
        BehaviorB( ) :: method( ... )
        )
})
```

In our sketch implementation, we can identify the **component** object that receives the **state** variables as a parameter. The first argument of the object is the name of the generic name of the agent that will be created when the component object is created. The **compRoleA** and **compRoleB** are the extensions of the component, and of course could be more. We could also create extensions on the extension. The methods **BehaviorA** and **BehaviorB** are the behaviors that every role adds. We can see that the argument of the **compRoleA** will be the state added by that role.

Once the structure is defined, the way of sending messages to the objects can be seen in the following code:

```
{

 aComponent : component("aState");
```

```
 sendo( aComponent, operation, []);

 aComponent2: sSendoBack(aComponent,
                     compRoleA, ["aStateA"]);
sendo(aComponent2, addBehaviorA, []);

aComponent: sSendoBack(aComponent compRolB, []);
sendo(aComponent,addBehaviorB, [])

}
```

What has been done first in the code is the creation of **aComponent** object, residing in a new agent. After that, we send to the **object** the method operation without parameters.

Then, **aComponent2** is created by using the synchronized callback *sSendoBack* to send the netmixin message **compRoleA** to **aComponent**. With that expression, we extend the object **aComponent**, placing the extension in a different agent. Then, we are distributing the cost of processing.

In the next expression the object **aComponent2** starts to play the Role A receiving in the next expression the message **addBehaviorA**.

The object **aComponent** changes its behavior by sending to himself the netmixin message **compRoleB**. Furthermore, the object changes its position, because the extension will be placed in a new agent, but we can still refer to the object with the same name, making the message sending completely transparent for the programmer. In the last expression of the code, the message **addBehaviorB** is sent asynchronously to **aComponent**.

Summarizing, we can see how a prototypical approach takes us to a simpler model for the solution. Moreover, using the distributed delegation, we can reduce the cost of processing the messages by distributing the extensions of the object. Thinking about a web application, we could place the role objects in the client site, increasing the speed on answering messages to the client.

5.2 Chat

One of the very well known case studies in distributed computing is the Chat Room. In this section we present and compare three different implementations. First, we study a class-based solution implemented with Java RMI, one of the most popular environment of development in class-based systems. Then, we present a simple implementation in Borg, but without using our mechanism, only with the facilities provided by the mobile multi-agent system. Afterwards, we review a solution implemented in our extension of Borg, showing how the concepts presented in chapter 4 can help to have a more organized design. Finally, we apply the advantages of mobility that Borg provides for agents, to show how powerful can be a solution implemented with a prototypical approach.

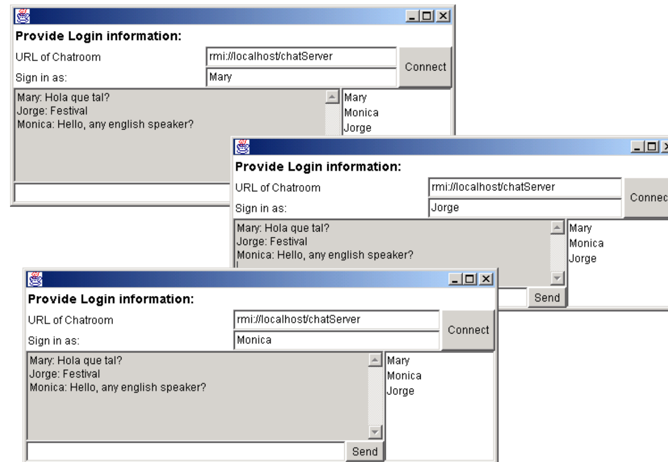

Figure 5.6: The Java RMI chat room.

5.2.1 Java RMI

To be impartial, we have chosen an implementation from the *Web*, taken from [Onl]. The design follows the rules of having right interfaces, and special types for sending the messages. It has a user friendly graphic interface that can be observed in the screenshot in figure 5.6, but we are not going to analyze it. We are going to concentrate in the design and implementation of the server and the client. Nevertheless, remark that the graphic user interface triggers the execution of sending messages from the client.

The chat design is based on the `ChatRoomImp` class that represent the server, and the `UserImp` class that correspond to the user. `ChatRoomImp` implements the Interface[2] named `ChatRoom`, which mainly provides the following methods:

- `enter(User user)` : log every user, adding it to a table of users.
- `getUserNames()` : get the list of users.
- `distributeMessage(Message message)` : distribute the message to all the users. This method use the getUserNames to know for whom the message has to be sent.

`UserImp` implements the Interface named User that only has two methods:

- `getName()` : get the nickname of the user. The graphic interface used it to send the message to the server, adding the nickname.
- `receiveMessage(Message m)` : receive a particular message *m*.

[2]Make the distinction between the graphic interface, and the Interface of Java to define the signature of the methods of a class

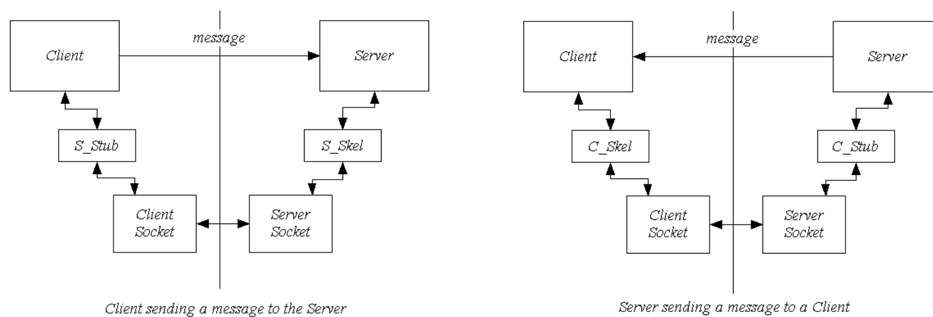

Figure 5.7: Components of the Java RMI chat room at Run Time.

The signature of the User does not have a way to send the message to the server; this is due to the fact that the graphic interface triggers this action directly to the server when user click on send button. This is just a decision of the particular design of the implementation. Not all the models do it like that. For instance, in the following implementations of Borg and our extension, we provide in the client a method to send the message to the server, mainly because we will not program the graphics user interface, due to we are going to use Borg graphic interfaces. Even though, we will see that the following two implementations have a similar structure, with a server having the list of registered users, and a way to distribute the messages to all of them. The clients perform the reception of the messages, and have a nickname as identifier. We are not going to review in details the Java RMI code because we considered Java well known. In any case, the whole implementation can be found in [Onl].

Once the code is compiled, as we explained in section 3.2.1, it is necessary to create the *skeletons* and *stubs* for the chat room and the user implementation. This is generated automatically using the RMI features. Now we just need to run the *rmiregistry* for the naming service, and we are ready to start the server and the clients. But let's count how many classes are needed to implement the chat. For the server side we need the Interface ChatRoom, its implementation ChatRoomImp and its skeleton and stub named ChatRoomImp_Skel and ChatRoomImp_Stub. For the client we need the Interface User, its implementation UserImp, with the correspondents UserImp_Skel and UserImp_Stub. In total, we need **eight classes** to communicate two objects, plus some other classes for typing messages and errors. It is even necessary that some of the classes correspondent to the server, like the skeleton and the interface, must be placed in the client, and vice versa. The components at runtime are depicted in figure 5.7.

The static typing system of Java can helps on having secure code, but renders much more complicated the structure for distributed and mobile systems. In the implementation of a chat, we can see that four classes, two skeletons and two stubs, are just needed for type checking in the communication between the ChatRoomImp and the UserImp. In the following sections, we will show how simple the design of chat can be using Borg and our extension of the language.

5.2.2 Borg

The implementation presented in this section is based on the primitive chat application of the Borg tutorial [Lab]. We have included nicknames to identify the users, and a method to send private messages to a particular user. There is no object orientation in this design, only a code base on agents and their capability to send asynchronous remote messages between them. There is no error-detection, nor detection of agents logging out, but the same features we analyzed in the Java RMI implementation, are presented in this solution: a server, clients logging in and sending messages to the chat room. In addition, as we already mentioned, we also provide a way of sending messages to a particular user.

The application consists in two pieces of code, one for the server and one for the client. The code for the server has to be present and loaded only in the server agent, meanwhile the client code, only belongs to the client agents. This is already an advantage over the Java RMI implementation, because both, the client and the server, can change their internal implementation at runtime, without notifying the other parties.

The Server

The code that has to be loaded at the server agent is the following.

```
{

users[10]:void; nicks[10]:void; nbrusers:0;

login(nickname, clientagent):: {
    nbrusers := nbrusers + 1;
    nicks[nbrusers] := nickname;
    users[nbrusers] := clientagent;
    chatsend("User" + nickname + " logged in", agentself)
};

chatsend(message, sender):: {
    if(sender=agentself, nick:"server",
      `get the nick of the sender
      for(i:1,((!is_void(users[i])) | (i<nbrusers)),i:=i+1,
            {tmp : users[i];
            if(tmp=sender, nick:nicks[i], void)} ) )
    `distribute the message
    for(i:1,((!is_void(users[i])) | (i<nbrusers)),i:=i+1,
        {tmp : users[i];
        msg:nick + ": " + message;
        tmp->chatreceive(msg)} )
    };
```

```
chat2nick(message, sender, receiver):: {
    if(sender=agentself, nickSender:"server",
      `get the nick of the sender,
      for(i:1,((!is_void(users[i])) | (i<nbrusers)),i:=i+1,
            {tmp : users[i];
            if(tmp=sender, nickSender:nicks[i], void)} ) )
    `get the reference to the receiver
    for(i:1,((!is_void(nicks[i])) | (i<nbrusers)),i:=i+1,
        {tmp : nicks[i];
        if(tmp=receiver, refReceiver:users[i], void)} )
    msg:nickSender + ": " + message;
    refReceiver->chatreceive(msg)
}

display("server loaded") }
```

The server uses two tables to register the nicknames and the references to user agents. Every time an agent loges in, it is added to the tables. The method `chatsend` is the equivalent to the method `distributeMessage` of the Java implementation, distributing the message to all the users of the table. Note that this message sending is asynchronous, then, the server does not wait until an agent receive the message to send the message to the next agent. This is a very important feature for distributed systems, because of the latency problems explained in chapter 3.

The method `chat2nick`, first get the nickname of the sender, then get the reference of the receiver, and finally send the message to the particular agent asynchronously. The way of registering the clients in tables, or the way of perform the distribution of the messages using the *for* instruction, can be modified at runtime without notifying the clients, and the chat will continue running. Of course, this have to be done respecting the signature of the methods. As we can see, the whole code is very simple.

The Client

The code that every client agent have to load in their address space is much simpler that the server one. It only has to provide two methods: one for receiving the messages sent by the server, and one to send the message to the server. We can see both methods in the following code. Note that the method for sending messages to the server could be avoided, because it is just a mask to hide the argument passing of the *agentself* variable. To check the difference, we are not going to mask the `chat2nick` function.

```
{
 chatreceive(message):display(eoln, message);
```

```
 chat(server, message):server->chatsend(message, agentself)
}
```

Once this code is loaded, the client can connect to the server as follows.

```
server : agent("machine/chatServer")
server->login("mynick", agentself)
```

First, we get the reference to the agent that serves the chat room, in this example placed on “machine/chatServer”. Then, we send the message login to the server, giving the nickname and the reference to our own agent. Then, to send messages to the chat room and to a particular nick, we do it like follows. Note the difference between the method that is masked (first), and the one that is not.

```
chat(server, "Hello, chinese anyone?")
server->chat2nick("liangJing", "ni hao!")
```

We can see that in both implementation we already reviewed, there are similarities in the general idea of building a chat room. But, as Borg is a language specially made for distributed computing, the design is much more simpler than the implementation using Java RMI, but still a little bit simplistic and unstructured. Figure 5.8 depicts the agent system of the chat implemented with Borg. We realize this a very small example to show how complicated it can become to maintain a big system, but in the following section, we aim to show how to structure a distributed program for a chat room using our extension of the language.

5.2.3 P-Borg

With the aim of giving an structured design to the chat room, we present in this section the implementation using P-Borg, already described in detail in chapter 4. The chat room consists in a server object with extensions. The extensions will be the clients of the chat, every one residing in a new agent. It is a different way to attack the problem, and we will see the advantages. Another important difference with the previous two implementations is that the whole code will be loaded only in one address space, where the server will be launched.

The implementation of the chat room using P-Borg is the following.

```
chat()::obj("theChat", {
```

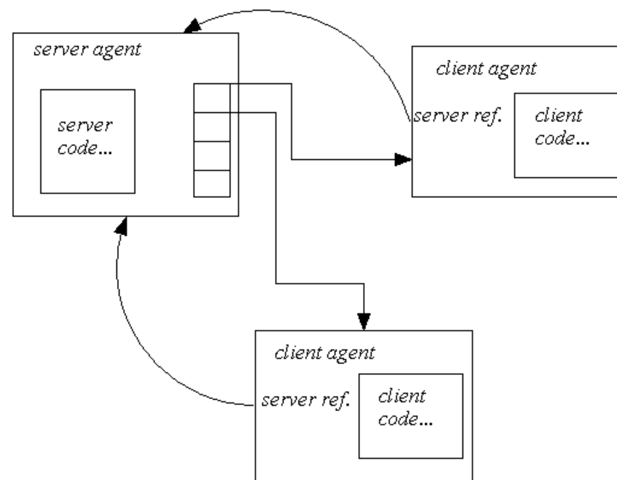

Figure 5.8: The Borg chat room.

```
  talk(msg)::method(sendo(us, showMsg, [nick, msg]));
  talkTo(to, msg)::method(sendo(yield, toMe, [self, to, msg]));

  client(nickname, address)::netmixin("theClient", {
    ui:agent(address);

    `Internal perfoming of messages
    showMsg(nick, msg)::
      method(sendo(ui, disp, [nick + "> " + msg + eoln]));
    toMe(from, to, msg)::method(
      if(to=nickname,
        sendo(self, showMsg, [from, msg]),
        "")
      )
    })
})
```

The primary object is *chat*, and once it is created it will resides in a agent named *theChat*. Create the object *chat* is the equivalent to launch the server, because the client extensions can start to be required since the object is created. Three methods are implemented in the server. The first one distributes the message to all the clients, and the second one sends a message to a particular nick. The third method is a *netmixin* to create client extensions. The structure of the chat is presented in figure 5.9.

To distribute the message to the user of the chat room, we are using the *us* variable. It

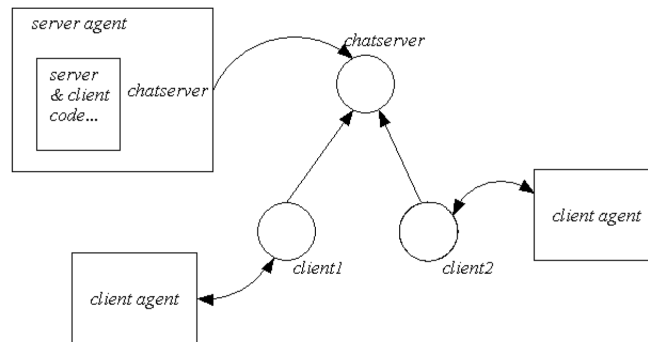

Figure 5.9: The structure of P-Borg chat room.

means we are sending the message `showMsg` to all the extensions, *i.e.* to all the clients, with the parameters `nick` and `msg`. Every client will perform the message as in the other implementations. What it is different here, is that we can use the advantages of the *us* variable, without having to distribute the message manually, and without using a data structure to register the users. As we only have one level of extensions, the variable *us* and *yield* have the same meaning. We use *yield* in the second method. Note that the method `talk` and `talkTo` are delegated to server.

To send a message to a particular nickname, we just send it to all the users, and then, every user identifies if the message is for him/her or not. We realize this way to implement the behavior generates unnecessary network traffic, because we are sending messages to users that are not the one for whom the message was destined. In this case it would be better identify the receiver of the message, and then send it.

As we already mentioned, creating extensions of the *chat*, we get a reference to the *client* agent. There are two important observations to say about the code of the client. First, as the client is a new agent, different to the one who creates the extension, we use *callbacks* to display the messages of the client. That is why we use the variable `ui` to refer the "owner" of the extension. Then, the methods `showMsg` and `toMe` can display the messages in the agent of the user of the chat. Second, we can say that the methods of the client can be divided in two parts: the interface for the use, and the internal methods to perform the messages.

What we call the interface for the user, corresponds to the two methods the user will use to send messages whether to the chat or to a specific nickname, `chat` and `chatTo` respectively. Note that the user interacts only with the client, and the client delegates the message to the server. We claim this presents a more organizer way to design the system. Even for the user, this structure represents better, from our point of view, the metaphor of typing messages in a client window.

The internal communication between the client and the server is implemented in the last two methods. As these methods are not supposed to be used by the users, we think about the possibility of having private and public methods. Where the public methods rep-

resents the messages that can be sent to an object just having a reference to it, meanwhile the private methods would be the methods accessible within the hierarchy of objects and its extensions, but not from outside.

Using the chat

A screenshot of the chat is presented in the figure 5.10. To launched the server of the chat room, some one creates a *chat* object with the following expression (lines that starts with colon ":" represents the result of the evaluation of the expression).

```
aChat:chat()
:<reference machine/theChat>
```

Then, users from different agents get a reference to the server to create a client extension.

```
server:agent("machine/theChat")
:<reference machine/theChat>
client1:sSendoBack(server,client,["bob","machine/Bob"])
:<reference machine/Bob>
```

Having the reference to the client object, in the example named `client1`, to send messages to the chat or to a nickname, we use the following expressions

```
sendo(client1, talk, ["hello, chilean anyone?"])
sendo(client1, talkTo, ["Miro", "Hola que tal?!"])
```

Concluding, we have presented the implementation of a chat room using P-Borg. From our point of view, as a design it is much more simpler to the implementation of Java RMI, and give more organization to the code of the implementation using Borg. It has the advantage that we can centralize the code in one server object, and then distribute the client extension. Then, the code is centralized but the objects and the execution are distributed.

Another advantage of our system is the usability of the variables *us* and *yield*, having an organized control of the extension of the primary object. What we did not exploit in this example, is the distributed delegation. In section 5.3 we present a better example to get an idea of the potential use of a distributed delegation, a clearer idea of the different overheads of the variables *us* and *yield*.

To finish with the chat example, we present the enormous advantage of having mobility in distributed systems, something that is hard to perform in class-based system, and that Borg does very simple and efficient.

Figure 5.10: The P-Borg chat room.

5.2.4 Using Mobility

One of the biggest merits of Borg is how it deals with mobility, offering a transparent way to do strong and weak mobility, and using the shortest way to communicate two agents wherever the agents migrate. We are going to add these features of Borg in the implementation of the chat with P-Borg. Our aim is to present that our extension is able to use all the good things of Borg, and also to remark that prototypes deal better with mobile systems than classes.

Imagine we have an expert user of the chat, which is generating more than 50% of the messages. A good idea to reduce network traffic would be migrate the server where this expert user is located. Then, the first communication between the user and the server will be very fast, and then the server will distribute the message to the other user as normal.

To avoid the explanation of an algorithm to control the amount of traffic, let's consider we have a *trafficGuard* to tell the server when and where migrate. We are only encapsulating the algorithm in this sort of guardian, nothing else. To use the guardian, we need to add three things to the server:

- A reference to the guardian object, to communicate with him.
- Every time we distribute a message, we tell the guardian who is the sender. Thus, the guardian can have the information of the amount of traffic that every user is generating. When, the guardian determines that the server has to move, it will send a message to the server saying where to move.
- We must provide a method that the guardian can use to tell the server where to move. It will be a simple method using the keyword *agentmove* provided by Borg, having the agent where to move as a parameter.

Referring to the code of the last section, what we need to add will looks like follows

```
chat()::obj("theChat", {

  guardian:trafficGuard(agentself);
  talk(nick, msg)::method({
      sendo(us, showMsg, [nick, msg]);
      sendo(guardian, newMsg, [nick])});
  migrate(agt)::method(agentmove(agt));

  client(nickname, address)::netmixin("theClient", {
    ...
  }
}
```

We can also continue adding algorithms to determine the best position for client objects and any kind of object we are using in the system, because every object is an agent that contains all the overheads provided by Borg. We can use all the load balancing and features for agents that Borg provides, because it is orthogonal to the organizational mechanism we provide.

5.3 A simple database

The next aim is showing the applicability of the distributed delegation, and trying to clarify the usage of the conceptual difference between the variables *us* and *yield*, that we have introduced in our system, we present in this section a simple database of students. The example can be related to the analysis of the Role Object Pattern, because in this example we are going to see the students as entities that can play different roles. The situation is the following:

> During the period of inscription, the students filled in a sheet with their personal information and some extracurricular activities they perform. In the sheet, students can put if they are amateur musicians, and furthermore, if they have a degree in any musical instrument. Another information is about sports, where they can indicate if they are regular sportsman/sportswoman, and indicating if they belong to the volleyball or basketball team of the university. All this information must be stored in a database that will be consulted via Internet.

With that information, we can create our database object with the correspondent extensions. The first level corresponds to the student objects. As a student can be a musician and-or a sportsman/sportswoman, then, we will have two *netmixins* for the students. Into the extension that corresponds to a musician, we will have another extension in case the

student has a degree in any musical instrument. Note that a musician, even when only one extension is defined, it can be extended two or more times in case the musician has a degree in two or more instruments. For the students who practice sports, we have two possible extensions in case they belong to a team of the university: volleyballPlayer or basketballPlayer. Note that these two extensions are a little bit different to the extension when the student has a degree in a musical instrument. If a student belongs to the volleyball team, will be extended to volleyballPlayer only once, meanwhile the musical instrument extension can be generated several times for the same musician student.

In this example, we will make two queries in the database. We are going to ask for the list of players of the basketball team, and then, we will ask the list of all the students that declared to be musicians. For the basketball team query we will access the yield of the structure, then, the *yield* variable will be useful. To get the musicians, we have to go only until the level of musician, without taking into account if they have or not a degree in a musical instrument. Then, the *us* variable will be useful to descent level by level into the structure of the data base.

The code presented below it has the minimum methods to answer the two queries of the example. In a normal system, every mixin should have more behavior and more state variables.

```
database()::obj("database", {

showMusicians()::method(sendo(us, musicians, []));
showBasketTeam()::method(sendo(yield, basketInfo, []));

student(name, email)::netmixin("student", {
  showInfo():method(
    display("Name  : ",name,eoln,"email : ",email,eoln));

  musicians()::method(sendo(us, musicInfo, []));
  musicInfo()::method(void);

  basketInfo()::method(void);

  showInstruments()::method(sendo(yield, instInfo, []));
  instInfo()::method(void);

  musician()::netmixin("musician", {
    musicInfo()::method(sendo(super, showInfo, []));

    instrument(inst)::netmixin("degree",{
      instInfo()::method({
        sendo(super, showInfo, []);
        display("Instr.: ", inst, eoln)})})
```

```
  });

  sportPlayer()::netmixin("sports", {
    sportInfo()::method(sendo(super, showInfo, []));

    basketPlayer()::netmixin("basket",{
      basketInfo()::method(sendo(super, showInfo, []))});

    volleyballPlayer()::netmixin("volley",{
      volleyballInfo()::method(sendo(super, showInfo,[]))
    })
  })
})

})
```

The two queries will be the result of sending the messages `showMusicians` and `showBasketTeam`. We can see that `showBasketTeam` send the message `basketinfo` to the yield of the structure. This message is only defined in the netmixin `basketPlayer`, that sends a message to the super to show the information of the student. All the other extensions will delegate implicitly the message `basketinfo` to the parent-link, that finally will stop at the student, where we see that `basketinfo` just return *void*. Then, all the rest of the extensions will return *void*.

In the case of `showMusicians`, the message `musicians` is sent to the *us* variable, in this case, to all the students. We can see that the method `musicians` in the student will send again a message to *us*, but now accessing the second level of the structure, *i.e.*, sport students and musicians. The musicians will finally show their info, and the rest of the students will delegate the message to the student extension, returning *void*, in a similar strategy of the basketball players.

Let's consider we fill the database with the following expressions:

```
{

db:database();

miro:sSendoBack(db, student, ["Miro", "miro@vub.ac.be"]);
miroMusic:sSendoBack(miro, musician, []);
miroMusicFlute:sSendoBack(miroMusic, instrument, ["flute"]);
miroMusicGuitar:sSendoBack(miroMusic, instrument, ["guitar"]);

isabel:sSendoBack(db, student, ["Isabel", "isabel@vub.ac.be"]);
```

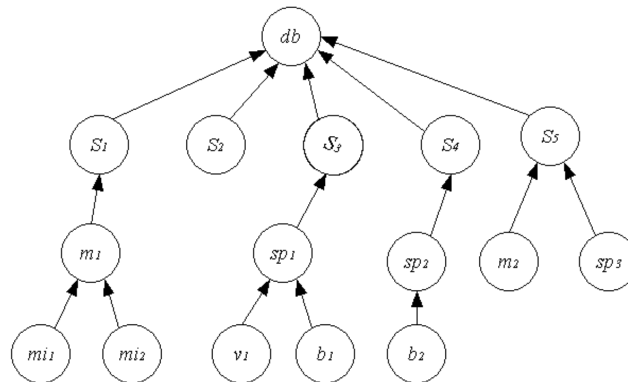

Figure 5.11: Data Base.

```
saartje:sSendoBack(db,student,["Saartje","saartje@vub.ac.be"]);
saartjeSport:sSendoBack(saartje, sportPlayer, []);
saartjeSportBasket:sSendoBack(saartjeSport, basketPlayer, []);
saartjeSportVolley:sSendoBack(saartjeSport,volleyballPlayer,[])

tchorix:sSendoBack(db,student,["Tchorix","tchorix@vub.ac.be"]);
tchorixSport:sSendoBack(tchorix, sportPlayer, []);
tchorixSportBasket:sSendoBack(tchorixSport, basketPlayer, []);

wolf:sSendoBack(db, student, ["Wolf", "wolf@vub.ac.be"]);
wolfMusic:sSendoBack(wolf, musician, []);
wolfSport:sSendoBack(wolf, sportPlayer, []);
}
```

All this information together will generate the tree shown in figure 5.11, where *db* is the database, s_i are students, m_i are musicians, mi_i musicians with a degree in a musical instrument, sp_i are sportsman/sportswoman, v_i are players of the volleyball team, and b_i players of the basketball team.

Why can't we send the message `showMusicians` to the yield, and why do we need to descent level by level using the variable *us*? Because we can get twice the same student. If the look at the figure 5.11, we can see that Miro plays two instruments, then, that student has two extensions in the yield that will reply two the query, duplicating the information. In the case of the basketball team, we are sure that there will be no double extension of basketball players, then, we can send the message `showBasketTeam` to the yield.

The queries and the answers are the following.

```
sendo(db, showMusicians, [])
```
:Name : Wolf

:email : wolf@vub.ac.be
:Name : Miro
:email : miro@vub.ac.be
:

```
sendo(db, showBasketTeam, [])
```

:Name : Saartje
:email : saartje@vub.ac.be
:Name : Tchorix
:email : tchorix@vub.ac.be
:

With this example, we showed the potential of the variables *us* and *yield*, and how useful distributed delegation between objects and extensions can be. Remember that all this objects can move through the network as agents in normal Borg programs, but with P-Borg we can give them a structure to maintain them and organize them. Our aim is not to implement distributed delegation in order to perform classic method-lookup over a network, our contribution applies the delegation and other features of the prototype-based theory to organize the mobile multi-agent systems.

5.4 Summary

We have presented three case studies in this chapter. First, we showed how our prototypical approach can be useful in the conceptual design of solutions, taking the Role Object Pattern as a point of comparison with the class-based programming. In that example we could see that modeling with prototypes results in a simpler model, and adding the fact that we are using distributed and mobile objects, our approach present big advantages when we want to apply the Role Object Pattern to web applications and distributed systems in general.

The second case is a very well known problem of distributed programming: a chat room. We present three different implementations to compare them, adding at the end the power of mobile agents. We presented a Java RMI implementation where we saw that the static typing obligate to have a hard structure to support type checking. Then, we saw how simple an implementation is in Borg, a language made to experiment with mobile computing. The third approach presented is using our extension of the language, adding structure to the Borg implementation. Finally, we showed that all the advantages of mobility provided by Borg, can be combined with our extension to have a well organized multi agent system, with big advantages over classes at the level of implementation, and reducing network traffic at runtime.

The third example presented demonstrates the usability of the variables *us* and *yield*, introduced in our approach to group the extensions of the objects, having a better way to access them. The example also exploits the distributed delegation, in this case to structure a solution to answer queries in a simple database.

The following chapter presents the conclusions of this thesis, and the future work.

Chapter 6

Conclusion and Future Work

6.1 Conclusions

We have presented in this dissertation a first approach to organize mobile multi agent systems, providing a conceptual language that have named P-Borg, a Prototype-based extension of Borg.

With P-Borg we have applied the techniques of the prototype-based theory to reason about how this concepts can help on organizing agents, obtaining interesting results. For instance, in the chat room we saw how the approach helps us to structure the agents in a hierarchical way. We also present an experimental way to model a distributed database, were our structure of objects with extension can be useful on the organization of the data, presenting how powerful prototypes can be.

But the design of the model has an important background that can be divided in two focuses: prototypes and distribution. In chapter 2 we introduce the concepts of prototype-based programming, starting with a philosophical point of view, presenting how strong is the prototype-based theory. Then, we present the conceptual differences with the class-based orientation, taking more time to explain the prototype-based concepts, presenting the advantages and disadvantages of them. As a first approach of organizing objects and delegation, we introduce the Split Objects, concluding the chapter with Agora and its mixin methods, important in the inspiration of our work.

The distributed and mobile computing theory was presented in chapter 3.We gave a brief introduction to the basic concepts that lead us to take distribution seriously. We continue the chapter presenting some of the most popular environment to develop distributed application, like Java RMI and CORBA. Then, we reviewed Obliq, a very well developed prototype-based language for distributed computing. At the end of the chapter we presented Borg thereby explaining its main features.

Chapter 4 was dedicated to describe our contribution. Considering we are extending a language, we start the chapter introducing macro functions, and giving some keys at the level of construction of Borg. Then, we present a first analysis where we concluded that objects could not share address space if we want them to live in a distributed environment,

moving around the network. We need autonomous and self-supported objects. Prototypes are self-supported, and for that reason we chose them instead of class-based objects, but we needed to adapt them to agents to be autonomous within the mobile environment.

Once defined P-Borg, we gave the semantics of the functions and concepts implemented in the language. Among those concepts, we introduce the pseudo variables ”us” and ”yield”, to help us in the aim of organization of multi-agent systems. These pseudo variables offer features that could be applied very well in a distributed scope, dealing with considerations of latency, partial failure and other basic concepts of distributed computing. At the end of the chapter we discussed some limitations and considerations to take into account to continue evolving the system.

To demonstrate the applicability and validation of the concepts introduced with P-Borg, we present three case studies in chapter 5. First, using the Role Object Pattern, we saw how helpful our approach can be to design general solutions. After the analysis of the Role Object Pattern, we can realize that conceptually prototypes appear as a very good alternative to model systems that need dynamic behavior. Nevertheless, the role object pattern show us that being clever in the decisions of design, class-based approach can finally deal with the dynamicity required, but with a sophisticated solution.

The second example goes more into the level of implementation, but without forgetting the main concepts at the moment of the analysis and comparison. This second example consists in the comparison of three implementations of a chat room, using Java RMI, Borg, and P-Borg, presenting the advantages of mobile code for distributed applications. With the chat room we also showed that all the well developed features of Borg can be incorporate to the prototype objects, allowing them to move around the network to reduce network traffic.

The last example is the design of a simple database, showing how the features of our extension can help to model and control a common source of information, for distributed queries.

We consider that the goals presented in the beginning of this dissertation are satisfactory achieved, but this is just a small contribution toward the primary aim of having a very well organized mobile multi agent system. Prototypes are indeed a helpful approach to structure agents, being able to migrate through the network using the features of agents. We believe that prototypes are promising in the aim of organizing mobile multi-agent systems, but as we already said, this is just a glimpse of a very long research track as we will see in the future work section.

6.2 Future Work

During analyses made in the chapters 4 and 5, we already gave some key ideas about future work. In this section, we are going to mention those ideas explicitly. From the list presented bellow, we consider the study of distributed split objects, and cloning with deep copy the two most interesting future works, with several consequences that potentially would generate new future work.

- Implement cloning: Until now, we are using the cloning operators provided by Borg to generate the objects and their extensions. But we are not providing a way to clone existing objects. In order to do that, we have to take the decision about the parent-link. We have two possibilities: the new agent gets a reference to the parent of the object we are cloning, or, we can also clone the parent object as it is explained in section 4.3.5.

- Code debugging: There is still some debug to do in the implementation of P-Borg, and adding new features we will probably need to modify some existing implementation. Initially, we would focus on the implementation of synchronized messages.

- Private in public method: Implementing some examples like the chat room application presented in section 5.2.3, we realized that objects have some methods that are only used for internal communication between them within the hierarchy of extensions, and some methods corresponding to the interface of the objects, received as messages sent from out of the hierarchy during the execution of the application. In this case, it could be useful to implement private and public method to ensure unwanted requested to the objects.

- Cloning with deep copy: To implement the distributed delegation, we forced the parent link to refer the agent where the parent object resides. But the cloning operator of Borg makes a deep copy of the dictionaries to the new agent to keep agent autonomous. Even when we are not saving space with this decision, it could be very useful to reduce networking traffic, but missing the sharing of common properties. It is also possible to think about having some methods distributed, and some others only locals, like proxies when methods are frequently requested. The aim of this future work is to reduce network traffic.

- Distributed Split Objects: In the first analysis made in chapter 4, we mention the possibility of using local mixin methods to implement distributed split objects. The idea is to have local extensions sharing the same address space than the primary object, as it is usually done with objects in non distributed computing. Then, these local extensions would be only accessible by the primary object, being this one, the only object possible to refer from out of the address space. Adding this kind of split objects to P-Borg, we can have distributed hierarchies composed by objects having local hierarchies, moving around the entire network, but with a well define structure!!

Bibliography

[BC95] K. Bharat and L. Cardelli. Distributed applications in a hypermedia setting, 1995.

[BFVD00] Werner Van Belle, Johan Fabry, Karsten Verelst, and Theo D'Hondt. Experience in mobile computing: The cborg mobile multi-agent system. Programming Technology Lab, Vrije Universiteit Brussel, 2000.

[BRSW97] D. Baumer, D. Riehle, W. Siberski, and M. Wulf. The role object pattern, 1997.

[BT79] B. J. Biddle and E. J. Thomas. Role theory: Concepts and research, 1979.

[Car94] Luca Cardelli. Obliq A language with distributed scope. Technical Report 122, 1994.

[Cha92] Craig Chambers. Object-oriented multi-methods in cecil. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 33–56, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.

[Ded01] Jessie Dedecker. Prototype-based languages and their programming idioms. Capita selecta, Vrije Universiteit Brussel, Ecole des Mines de Nantes, 2001.

[D'H] Theo D'Hondt. The pico website. *http://pico.vub.ac.be*.

[DM00] Theo D'Hondt and Isabel Michiels. Combating the paucity of paradigms in current oop teaching. Programming Technology Lab, Vrije Universiteit Brussel, 2000.

[Hut96] Norman C. Hutchinson. An emerald primer, 1996.

[Ken] Elizabeth A. Kendall. Role models for agent system analysis, design, and implementation.

[KM96] Bent Bruun Kristensen and Daniel C. M. May. Activities: Abstractions for collective behavior. *Lecture Notes in Computer Science*, vol. 1098, 1996.

[Lab] Prog Lab. The borg website. *http://borg.rave.org*.

[Lie86] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 21, pages 214–223, New York, NY, 1986. ACM Press.

[Meu98a] Wolfgang De Meuter. Agora: The story of the simplest mop in the world - or - the scheme of object-orientation. 1998.

[Meu98b] Wolfgang De Meuter. Agora98 language manual. 1998.

[Mic] Sun Microsystems. The java remote method invocation RMI homepage. *http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html*.

[NCRB+87] Hutchinson Norman C, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The Emerald programming language. Technical Report 87-10-07, Seattle, WA (USA), 1987.

[Onl] Internet Computing Online. *http://www.computer.org/internet/v5n1/rmitut.htm*.

[Rie98] Dirk Riehle. Bureaucracy. In Robert Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, pages 163–185. Addison Wesley, 1998.

[Sie00] J. Siegel. *Corba 3 Fundamentals and Programming*. John Wiley & Sons, January 2000.

[SLS+94] R.B. Smith, M. Lentczner, W.R. Smith, A. Taivalsaari, and D. Ungar. Prototype-based languages: object lessons from class-free programming (panel). In *Conference Proceedings (Portland, Oregon, October 23-27) OOPSLA'94*, volume 29, pages 102–112. ACM, 1994.

[SLU89] 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.

[Ste87] Lynn Andrea Stein. Delegation is inheritance. In Norman Meyrowitz, editor, *Proceedings OOPSLA'87 Coference (Orlando, Florida, October 4-8)*, volume 29, pages 138–146. ACM, 1987.

[Tai96] Antero Taivalsaari. Classes vs. prototypes - some philosophical and historical observations. april 1996.

[US87] D. Ungar and R. Smith. Self: The power of simplicity. In Norman Meyrowitz, editor, *Proceedings OOPSLA'87 Coference (Orlando, Florida, October 4-8)*, volume 22, pages 227–241. ACM, 1987.

[WWWK97] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. In *Mobile Object Systems: Towards the Programmable Internet*, pages 49–64. Springer-Verlag: Heidelberg, Germany, 1997.

Appendix A

P-Borg Implementation

```
{

comment(IdontCare()):void;

 obj_slots(obj):obj[3];
 dct_name(dct):text(dct[1]);
 dct_val(dct):dct[2];
 dct_next(dct):dct[3];

ref_tag:1;
   comment("This is the tag of a reference to a variable");
app_tag:2;
   comment("This is the tag of a function application");
msg_tag:4;
   comment("This is the tag of a message expression");
 comment("There is a primitive -tag- to get the tag");
 comment("There is a primitive -make- to make a tree");

 comment("The types to send messages to agents");
 sendoType::1;
 sSendoType::2;
 comment("do not use this sSendoType.");
 comment("As we have sSendoBack, having this one it will be like");
 comment("making distintion between procedures and functions");
 sendoBackType::3;
 sSendoBackType::4;

comment("Types of delegation"); selfType::1; yieldType::3;
```

```
equivalent@args:~@args;

obj( _where_ , exps( _where_ )):{
   comment("We will build a new tree for the object");
   beginTree:make(app_tag);
   beginTree[1]:=begin[1];
   beginTree[2]:=[
     comment("Install the super variable"),
     read("super:void"),
     comment("To get the reference to us"),
     read("we:empty"),
     read("addNetMixin(ag)::we:=cons(ag, we)"),
     read("cleanUs()::we:=empty"),
     comment("add the body of the object"),
     exps[3],
     comment("cloning the object placing it into a agent"),
     read("me:clone2agent(_where_)"),
     comment("initializing us"),
     read("me->addNetMixin(me)"),
     comment("return the reference to the clone agent"),
     read("me") ];
   exps[3]:=beginTree;
   comment("we get a lambda function with the object");
   lam(_where_):exps(_where_);
   comment("we apply the lambda function");
   lam@[_where_]};

method(exps(self,me, us, yield)):{
   lambda(Aself,Ame, Aus, Ayield):exps(Aself,Ame, Aus, Ayield)};

netmixin(_where_, exps(self,me, us, yield, _where_)):{
   comment("We will create a Tree");
   beginTree:make(app_tag);
   beginTree[1]:=begin[1];
   beginTree[2]:=[
     comment("Install the super variable"),
     read("super:agent(text(agentself[2]))"),
     comment("To get the reference to us"),
     read("we:empty"),
     read("addNetMixin(ag)::we:=cons(ag, we)"),
     read("cleanUs()::we:=empty"),
     comment("add the body of the object"),
     exps[3],
```

```
      comment("cloning the object placing it into a agent"),
      read("me:clone2agent(_where_)"),
      comment("updating super.us"),
      read("super->addNetMixin(me)"),
      comment("initializing us"),
      read("me->addNetMixin(me)"),
      comment("return the reference to the clone agent"),
      read("me") ];
   exps[3]:=beginTree;
   lambda(Aself,Ame, Aus, Ayield):
      exps(Aself,Ame, Aus, Ayield, _where_)};

BApply(me,m,args,self, caller, stype):{
   comment("arguments filled in");
   lam:m@args;
   blabla("going for some self now");
   res:lam@[self,me, "us", "yield"];
   if(stype=sendoType,
      res,
      if(stype=sendoBackType,
         send(caller, res),
         if(stype=sSendoBackType,
            ssend(caller, res))))
   };

hasM(dct,m):{
   comment("looking for m within dct");
   if((is_void(dct_name(dct))) |
      (dct_name(dct)="[frame]") |
      (dct_name(dct)="[dctname]"),
      false,
      if(dct_name(dct)=m,
         dct_val(dct),
         hasM(dct_next(dct),m)))};

hasSuper(dct):{
   comment("lookig for super within dct");
   if((is_void(dct_name(dct))) |
      (dct_name(dct)="[frame]") |
      (dct_name(dct)="[dctname]"),
      false,
      if(dct_name(dct)="super",
         !is_void(dct_val(dct)),
```

```
          hasSuper(dct_next(dct))))};

hasWe(dct):{
   comment("lookig for we within dct");
   if(dct_name(dct)="we",
      dct_val(dct),
      hasWe(dct_next(dct)))};

send2Yield(m,args,self, caller, stype)::{
   comment("me becomes the dictionary of agentself");
   me:obj_slots(agentself);
   theWe:hasWe(obj_slots(me));
   if(equivalent(cdr(theWe), empty),
      {comment("I am one of Them");
      delegate(m,args,self, caller, stype)
      },
      while(!equivalent(cdr(theWe), empty),
         {comment("Broadcast to my extensions");
         blabla("BC from ", self,eoln);
         delegate2agent(
            car(theWe), m, obj_slots(me),
            args, self, caller, stype,
            yieldType);
         theWe:=cdr(theWe)
         })
      )
};

delegate(m,args,self, caller, stype):{
   comment("me becomes the dictionary of agentself");
   me:obj_slots(agentself);
   res:hasM(obj_slots(me),m);
   if(equivalent(res,false), {
      comment("no method... looking harder...");
      res:=hasSuper(obj_slots(me));
      if(equivalent(res,false),
         error("Message not understood "+m), {
         if((stype=sendoType) | (stype=sendoBackType),
            delegate2agent(me.super, m, obj_slots(me),
               args, self, caller,
               stype, selfType),
            {
            comment("if is synchronized, I become the caller");
```

```
            delegate2agent(me.super, m, obj_slots(me),
               args, self, self,
               stype, selfType);
            res:=srecv(me.super, any);
            ssend(caller, res)
            }
         )
   } ) }, {
   comment("ahaa... found... going to apply it");
   BApply(me,res,args,self, caller, stype)})};

delegate2agent(rec, app, dct, args, self, caller, stype, dtype):{
   unevaluated_args:args;
   evaluated_args:void;
   comment("going to evaluated the arguments");
   if (is_symbol(unevaluated_args),
      evaluated_args:=eval(unevaluated_args,dct),
      {
      siz:size(unevaluated_args);
      tbl[siz]:void;
      evaluated_args:=tbl;
      idx:1;
      while(idx<=siz,
          {
          evaluated_args[idx]:=eval(unevaluated_args[idx],dct);
          idx:=idx+1
          })
      });
    comment("Preparing message to be send");
    if(dtype=selfType,
        refDel:delegate[1],
        if(dtype=usType,
            refDel:send2YouAndYours[1],
            refDel:send2Yield[1]));
    comment("arguments of delegate");
    argsDel[5]:void;
    argsDel[1]:=app;
    argsDel[2]:=evaluated_args;
    argsDel[3]:=self;
    argsDel[4]:=caller;
    argsDel[5]:=stype;
    tosend:async_msg(rec, refDel, argsDel);
    comment("call in the caller's environment =>");
```

```
    tocall:eval(quote(=>),dct);
    tocall(rdc_agtname(rec),tosend)};

sendoFilter(rec, nam, dct, args, stype)::{
   caller:agentself;
   if(!is_ref(rec),
   {
      comment("The receiver is self, us or yield");
      if(is_text(rec),{
         comment("The receiver is us or yield");
         me:obj_slots(agentself);
         theWe:hasWe(obj_slots(me));
         if(equivalent(cdr(theWe), empty),
            delegate2agent(car(theWe), nam, dct,
               args, car(theWe), caller, stype,
               selfType),
            if(rec="us",
               while(!equivalent(cdr(theWe), empty), {
                  delegate2agent(car(theWe), nam,
                     dct, args, car(theWe), caller,
                     stype, selfType);
                  theWe:=cdr(theWe)
                  }),
                  while(!equivalent(cdr(theWe), empty), {
                     delegate2agent(car(theWe), nam, dct,
                        args, car(theWe), caller, stype,
                        yieldType);
                     theWe:=cdr(theWe)
                  })
            )
         )},{
         comment("Calling Self");
         delegate2agent(dct_val(rec),nam, dct,
            args, dct_val(rec), caller, stype,
            selfType)
         })
      },
      {
      comment("sending the message to the agent");
      delegate2agent(rec, nam, dct, args, rec,
         caller, stype, selfType)
      })
};
```

```
sendo(rec,m(),args):{
   nam:text(m[3]);
   dct:m[4];
   sendoFilter(rec, nam, dct, args, sendoType);
   ""
};

sendoBack(rec,m(),args):{
   nam:text(m[3]);
   dct:m[4];
   sendoFilter(rec, nam, dct, args, sendoBackType);
   ""
};

sSendoBack(rec,m(),args):{
   nam:text(m[3]);
   dct:m[4];
   sendoFilter(rec, nam, dct, args, sSendoBackType);
   srecv(rec, any)
}

}
```