Using the Prototype-based Programming Paradigm for Structuring Mobile Applications

Jessie Dedecker*and Wolfgang De Meuter
(jededeck | wdmeuter)@vub.ac.be
Vrije Universiteit Brussel
Department of Informatics
Pleinlaan 2 - 1050 Brussels - Belgium

October 27, 2002

Abstract

Many mobile agent systems have been developed in the last decade in the form of APIs for existing languages or as brand new languages. Although such mobile agents have many benefits, they have not yet conquered the internet. This is partly because programming such mobile agents is not a straightforward task. There is a need for languages that better structure such mobile agent applications. In this position paper we want to advocate the use of prototype based paradigm as a basis for programming such mobile agents.

1 Introduction

Many mobile agent systems have been developed in the last decade in the form of APIs for existing languages or as brand new languages. Mobile agent systems are autonomous programs that can decide to move from one machine to another in the network.

Mobile agents have many interesting properties [CHK97] such as support for weak clients (i.e. mobile phones), support for disconnected operations, robust remote interactions, ... Despite these benefits, they have not yet conquered the internet. This is partly because designing and programming such mobile agents is not a straightforward task. There is a need for languages that allow the programmer to better structure such mobile agent applications.

2 Why Prototypes?

The object-oriented paradigm is usually associated with class based languages. There is an alternative based on prototypes where all objects are self sustaining, so they do not depend on a class for their behavior. Objects are created 'ex nihilo' (out of nothing, that is by putting attributes together) or by cloning existing objects rather than instantiating them from a class. In class-based languages objects share their behavior via the class, while in prototype based languages objects can share both behavior and state via *delegation*. This kind of delegation should not be confused with the delegation pattern, better known as the deco-

*Research Assistent of the Fund for Scientific Research - Flanders, Belgium (F.W.O.)

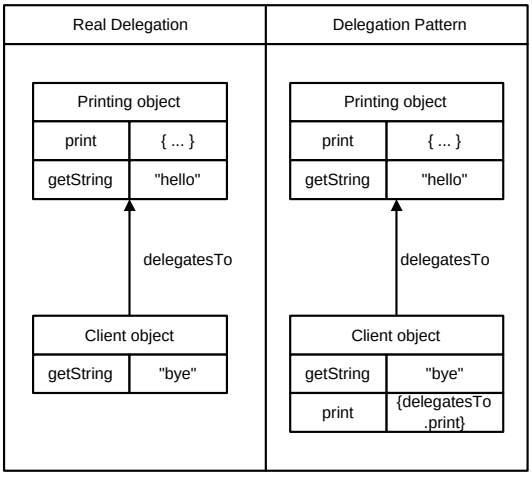

Figure 1: Real Delegation vs. Delegation Pattern

rator pattern, from [GHJV94]. We illustrate the difference with following example: Consider a printing object that puts some string on the screen when the message print is received. The string that is to be printed on the screen is retrieved via self.getString in the method body associated with the print message. One difference by looking at the code of the objects is that in the delegation pattern the message print is manually invoking a send of the message print to its parent, while this happens transparently in real delegation. Another more important difference is the outcome of sending the message print to the *client object*. In real delegation the string *bye* will be printed on the screen, while with the delegation pattern the string *hello* will be printed on the screen. The reason for this is that with real delegation the *self* is bound to the sender of the message, while with delegation pattern the *self* is rebound to the receiver of the message with each message that is sent.

In the remainder of this section we shortly discuss some of the advantages.

2.1 No Classes

A prototypical object does not depend on a class definition to find its behavior. In class-based mobile agent systems this has a number of disadvantages:

- whenever an object decides to migrate we need to decide whether we will migrate the class definition with the object. Leaving the class definition behind has some disadvantages:
  - severe performance penalties, because each time the migrated object receives a message it needs to use the network to obtain the behavior associated with that message.
  - partial failures, when the network connection between the class object and the migrated object fails or the machine with the class object fails we cannot process anymore message for that object since it cannot access its behavior.
- when we decide to migrate the class definition with the migrating object we get the same problems if multiple instances exist on the old machine, unless we decide to migrate all the instances.
- making copies of the class object makes it impossible to update the class object with new behavior at run-time (as the changes will not be propagates to all of its instances). This restricts the adaptability of the software to new environments, which is a must for mobile agents that are put into unknown environments. Also the class variables become a problem when we start replicating classes on different machines.

In the discussion above we have omitted the problem of inheritance, which would complicate the problems further. When objects do

not depend on classes for their behavior we have less problems migrating the objects into new environments.

2.2 Cloning

We already discussed that new objects in prototype based languages are created 'ex nihilo' or by cloning. Cloning is an interesting operation for distributed applications, because it allows to create replica's with a simple command. Replica's are used to overcome partial failures. Such partial failures are an inherent problem to distributed computing and each serious distributed applications should consider them.

There exist several variations for the cloning of objects:

- shallow clone:
  takes a copy of the state of the object and shares the behavior with the object from which it was cloned by means of delegation.

- deep clone:
  takes a copy of both the state and the behavior of the object.

Shallow clones would introduce many of the problems we discussed as with class-based agent systems, although in some cases they can be interesting (i.e. when groups of objects are always migrating together). Deep clones on the other hand create fully independent objects that can migrate freely between different machines. We think that the cloning operators are a good basis to start with for mobile agent systems, but they should be further extended so that different types of cloned objects can be created depending on some parameters.

2.3 Reflection and Meta-Architectures

In a world where everything is represented as an object, classes are also represented as objects such as in Smalltalk. These class objects also need a class object to exist, these class objects in turn again depend on other class objects to exist, and so on... This phenomenon is called infinite regression and makes the meta-architecture of class-based languages more difficult to grasp. Prototype based languages have meta-architectures that are easier to understand, because we do not have infinite regression as objects do not depend on classes, caused by the classes.

Mobile agent systems could benefit from a good and easy-to-use meta-architecture, because they can be used to separate concerns in mobile agent applications. We could for example make use of the meta-architecture to separate the mobility concerns or replica management from the other code, by implementing these concerns at the meta-level.

2.4 Dynamic Typing

Many of the prototype-based languages are dynamically typed. This is important when we are programming agents for open distributed systems, such as the internet, where agents have to interact with unanticipated other agents. The reflective properties of the language can be used to discover the interfaces of new environments and new agents that an agent encounters when migrating.

3 Conclusion

We believe that these four ingredients that are present in most prototype-based object-oriented languages form a good foundation to program and organize mobile program. In future work we will adapt and extend the

paradigm with language constructs that enhance the structuring and development of such programs.

References

[CHK97] D. Chess, C. G. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? In J. Vitek and C. Tschudin, editors, *Mobile Object Systems - Towards the Programmable Internet*, Lecture Notes in Computer Science, pages 25–47. Springer-Verlag, Berlin Germany, 1997.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.