Object-Oriented Programming on the Network

Jim Waldo

Sun Microsystems, Inc., 1 Network Drive, Burlington, MA, 01803,U.S.A.
jim.waldo@sun.com

Abstract. Object-oriented programming techniques have been used with great success for some time. But the techniques of object-oriented programming have been largely confined to the single address space, and have not been applicable to distributed systems. Recent advances in language technology have allowed a change in the way distributed systems are constructed that does allow real object-oriented programming on the network. But these advances also change some of our most basic conceptions about the relationship between processor and code, and what it is that constitutes a computer. We will argue that a new computing architecture, based around the ideas of the network and full object-orientation, will soon become the dominant computing architecture, allowing us to tie together large numbers of devices but requiring that we think and design in entirely new ways.

1 Object-Oriented Systems

Over the past fifteen years, object-oriented programming has moved from the research lab to the mainstream. The use of object-oriented techniques is rarely questioned these days; organizing a program as a group of interacting components, each of which is treated as an object, is now the default way of designing any program of any size. To do otherwise is sometimes necessary, but that necessity must be justified; the default way of organizing a program or system of any size is to use object-oriented techniques.

Of course, different people have different notions of what it is for a program or sysem to be constructed along object-oriented lines. To program in an object-oriented fashion is more than using a language that labels itself as an object-oriented language. All of us have seen many C programs with little or no object-orientation written in either C++ or Java. Some of the better object-oriented programs I have had the pleasure of reading have been written in languages that are not themselves considered object-oriented. Real object-orientation is a design philosophy, not something that is automatically gained by choosing a language.

Given the wide variety of what is meant by „object-oriented programming“, perhaps I should start by heeding the advice of Humpty Dumpty in *Through the Looking Glass*[1] and say what I mean when I use the term. The object-oriented approach to programming rests, I believe, on a small number of principle.: The first of these is the principle of abstraction- that the actual representation of an object is an accidental property of the object, with the essence of the object being shown by the operations that can be performed on that object.

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 441-448, 1999.

This first principle gets stated in a number of different ways. Sometimes this principle is stated as the independence of an object's interface from the implementation of the object. Other times it is stated as a principle of data hiding. By whatever name, it is at the base of object-oriented programming. It is by following this principle that we insulate ourselves from changes in the underlying data representation of an object, allow alternate implementations of the same interface, and compartmentalize our code to make it more manageable and, we hope, more stable.

The second principle underlying the power of object-oriented programming is the binding of behavior with data. This may seem like a variant of the abstraction principle described above, but in fact it is quite different. The ability to associate code with an object (or a class of objects) is necessary for abstraction, but goes far beyond what is needed for abstraction. Since an object can exhibit arbitrarily complex behavior, and since that behavior can be changed over time, the binding of object and behavior does more than just abstract the information contained by the object. In a real sense, what this association allows is a distinction between what an object does and how it goes about doing it. All a user of an object needs to know is the what of an object; the how is a private matter to the object itself, which can change without the client changing.

The third principle underlying the power of object-oriented programming is what I will refer to as polymorphism. This is often described as the ability of an object to have multiple forms. I take it to be a somewhat more interesting principle, centering around the ability to describe an object in terms of the necessary conditions on the object. When a client of an object wishes to describe the kind of object in which it is interested, it needs only say what the necessary conditions are for the object. Beyond those conditions, the object can vary in any way at all. This is more than an object having many forms. It is also the ability of users of an object to only need to specify one of the forms of the object, and allowing variation beyond that particular form.

These three principles are, I believe, all that there is to the core of object-oriented programming. There are lots of other characterizations of the subject (having to do, for example, with reuse, or design patterns, or whatever), but they all come down, in the end, to some variant or combination of these three.

While these principles and their statement may be somewhat idiosyncratic (and are also, no doubt, imprecise and perhaps incomplete), they are not new. These kinds of points have been made about object-oriented programming for years. So why do I bring them up?

I bring them up because my work has been in the combined fields of object-oriented programming and distributed systems. And in distributed systems, most of what gives power to object-oriented techniques has, until quite recently, been unavailable. While there has been lots of talk about object-oriented distributed systems, in fact such systems have been impossible under the usual assumptions that have governed distributed computing.

2 Distributed Systems

Distributed systems would seem to be a natural area for the application of object-oriented techniques. A distributed system, roughly speaking, is any system containing multiple computers that communicate over a network and share state[2]. The processes that run on these computers can generally be thought of as objects, each of

which has an interface that is used by other objects on other machines to make requests. Since the processes are separated by the network, there is already a hard and enforceable abstraction boundary.

Indeed, distributed systems often describe themselves as object-oriented, pointing out that they require interfaces, do not allow access to internal state, and must be constructed in a way that insures that the implementation of a distributed object can be changed without the client using that distributed object through the interface ever knowing. When viewed from enough distance, this claim seems reasonable. The objects tend to be large, and they tend to be rather static, but they are objects none the less, defined by an interface and accessed through that interface.

However, if we look at the kinds of interfaces that are presented by the objects in most distributed object systems, they have a decidedly non-object-oriented flavor. The information passed between these distributed objects is limited in rather sever ways. In most common distributed object frameworks, the information that can be passed from one object to another is limited to a small number of primitive data types, or references to other distributed objects, or structures made up of these types and references. One doesn't see objects being passed from one place in a distributed system to some other place in that system. And this seems strange, because if you look at examples of object-oriented interfaces in non-distributed systems, the vast majority of the information passed from one object to another object as a parameter or a return value in a method call is another object.

The reason for this oddity is both simple and deep. When doing distributed computing, there are a certain set of assumptions that have been made that define the environment in which distributed computing takes place. Because of these assumptions, the information that can be passed in such systems is severely limited.

The assumptions that I am talking about revolve around the presumption that a distributed system is made up of heterogeneous parts. The processors in a distributed system may be of different types, with different instruction sets and different data layouts. The operating systems on the machines built around those processors may be different. The languages that are used in implementing the objects that are communicating may well be different, and have been assumed to be compiled languages, that produce object code that is specialized for the instruction set of the processor and, perhaps, operating system in which the program is going to run.

Given this set of assumptions, it is rather remarkable that any distributed system has ever run at all. In an environment this hostile, getting any information from one process to another on some other machine is nothing short of miraculous. And the miracle has occurred because, no matter what sort of object-oriented veneer we have been able to put on our distributed systems, those systems have been able to work because at the most fundamental level they have not been object-oriented. Indeed, at the most fundamental level, those systems have been ways of defining the lowest level wire protocols imaginable.

To see this, all we need to do is to reflect on what really goes on in a system like CORBA[3] or DCOM[4]. Interfaces are defined in an interface definition language, where method calls are defined in terms of the primitive data types, object references, and structures of these entities that are passed back and forth for the interface. These interfaces are then compiled, for any given implementation language, with the result being stub and skeleton files. Once these are augmented with code that needs to be provided by a human programmer, they can be compiled for the target operating system and processor.

The result is object code for the particular processor and operating system that will take a call from a client, translate that call into the wire protocol, and sent those bits across the wire. Once received, the skeleton will translate the known bits into something that can be understood in the context of the receiver, make a call to the server, and then translate any results into the wire protocol that can be sent back to the original client. These known bits on the wire can be reformulated into something known by the client, and the call if finished.

While this is a great convenience over forming the bits on the wire by hand, it is hardly object-oriented. The information passed between the participants cannot change without both the client and server being updated simultaneously, since each must know exactly what is transmitted between the two parties. There is no notion of knowing a minimal set of conditions about what is transmitted; what is needed is exact knowledge. There is isolation, but the isolation is bought at the price of a static interface, hard to update or change. There is certainly no notion of behavior being sent from one participant in the distribute system to another.

Given the assumptions for distributed computing, it can hardly be otherwise. Since there is no single implementation language that can be assumed, all that can be passed from one distributed object to another are things that are common to all programming languages. Given the assumption of compiled code and different processors, only data can be handed from one distributed object to another, since code must be assumed to be local to a particular machine. While we can model the network as a whole as a set of cooperating objects, the way in which those objects cooperate and communicate is decidedly non-object-oriented.

3 Adding Objects to the Network

The advent and wide adoption of the Java(tm) programming language[5] and environment[6] has provided an opportunity to examine what can be done when the basic assumptions of distributed computing no longer hold. By assuming that the Java environment is available on every member of a distributed system, we get a very different network environment in which to develop our object-oriented techniques.

To begin with, the Java environment provides a homogeneous layer on top of the heterogeneity of the distributed system. Rather than thinking in terms of particular processors and operating systems, the Java environment provides a virtual machine that is (more or less) the same everywhere. While this property has generally been understood as giving the ability to install the same code on different processors, in the distributed system context this should be understood as offering the same system everywhere.

This homogeneity allows the second great advantage of the Java environment as a distributed computing environment- the ability to move code from one machine to another. The portable binary code that Java provides means by which behavior can be moved around the network, which in turn allows real objects (both code and data) to become network citizens.

Of course, there are other ways of providing an environment that is both homogeneous and has portable binary code- if all of the machines on a network are of a single processor type, running the same (or compatible) operating systems, then code can be moved around. The additional aspect of the Java environment that is as important as the other two is that the language is safe and the code that is moved (and dynamically

loaded) can be verified to follow the rules imposed by the language. This notion of safety is something that must be a part of the general distributed computing environment, and cannot be provided by simply having the same machine type and operating system everywhere.

The combination of a homogeneous environment, portable object code, and verifiable safety enabled Applets, which in turn brought the Java language to the attention of most programmers. By supplying objects that implemented a well-known interface, and making sure that browsers for the World-Wide Web knew how to recognize such objects and invoke the appropriate methods in that well known interface, Java allowed the introduction of active content to the world wide web.

Applets were the first application of object-oriented principles to a large scale network. By allowing different implementations to be offered for the same (known) interface, and moving that implementation into the browser that made the calls to that interface, Java Applets allowed alternate implementations of a known interface to be used to extend existing programs in interesting ways.

By taking one more step, the application of object-oriented techniques to the network could be made more general. By rejecting the usual assumption that the programming language in which a distributed object doesn't matter and instead assuming that all of the objects in a distributed system are written in the Java programming language (and running in the Java environment), the Applet approach could be generalized to any distributed program.

This ability was introduced into Java at JDK 1.1 with the introduction of the Java Remote Method Invocation system (RMI)[7]. RMI allows Java objects to be passed from one Java virtual machine to another, even when those virtual machines are located on different physical machines. Further, RMI will pass an object by its true type, not the type that is declared in the method signature of the remote method. This allows the passing of subtypes to methods declared to use a supertype. If the receiving virtual machine does not have the code associated with the actual class of the object that it receives, the code for that class is downloaded, verified, and dynamically loaded into the receiving virtual machine.

This is done for both local and remote objects. For a local object, this means that any calls made to a new subtype of thedeclared type that is passed in to the address space of the receiving machine will invoke the new behavior (if any) associated with the subtype. This allows many of the usual object-oriented patterns, previously confined to a single address space, to be used in a distributed system. This also means that a program can be updated dynamically, by receiving new implementations of classes as objects associated with those new implementations become available.

For remote objects, the case is somewhat different but just as important. The code that is received for a remote object is, effectively, the stub for that object. This stub will reflect all of the remote interfaces supported by the remote object. Using the standard Java type mechanisms and reflection, these interfaces can be discovered and used. Further, since the stub code can be downloaded, there is no need for the client to ever have to worry, either directly or indirectly, about the actual wire protocol used to communicate with the remote object. This protocol becomes a private matter between the remote object and the stub that is provided by that object. If the protocol changes, the client never needs to know--it simply makes the same calls to a local object (the downloaded stub) that has a different implementation.

This is the base functionality that has allowed the third stage of object-oriented network programming based on the Java environment, the Jini(tm) system[8]. The Jini architecture is based on the ability to move code from one machine to another, and an

exploitation of the Java type system as a way of identifying services that can be used by other members of the distributed system.

The Jini infrastructure extends the basic Java platform (including RMI) by adding a component, called the Lookup Service, that allows services to advertise themselves, and a simple protocol that allows these services and clients wanting to find a service to first find a Lookup Service. By using this protocol, joining a Jini network can be virtually automatic. A client or service wishing to find a Lookup Service sends out (via multicast) a known packet. Any Lookup Service receiving this packet will reply (to an address contained in the packet) with an implementation of the interface to the Lookup Service itself.

Once this object has been received (with, perhaps, the code for the particular implementation being downloaded), an object wanting to offer a service registers with the Lookup Service. This registration consists of handing the Lookup Service an object that can be used to access the service, along with an optional set of attributes that can be used to identify the service in ways other than its Java type.

A client wanting to use a service finds a Lookup Service using the same discovery mechanism, and also receives an object that implements the Lookup interface. Rather than registering a service, however, a client will request a service. It does this by specifying the Java type of the service that it wishes to use, along with any possible attributes that might make it want to choose between different services that implement the same interface.

On receiving such a request, the Lookup Service will return a copy of the object placed in it by the service (assuming that the object implements the interface that was requested by the client). This object may actually implement a subtype of the interface requested, or other interfaces in addition to the one requested. This can be discovered by the usual Java type and reflection mechanisms. The important thing is that, when the client receives the object, any code that might be needed by the client for the object implementation will be downloaded into the client if it is not there already.

The effect this has is that the client contacts the service using code supplied by the service. How this contact is done is up to the original service; all the client needs to know is the interface (indeed, the minimal interface) needed to talk to the service. Different implementations of the same service can use different communication mechanisms, which are encapsulated in the implementation of the object obtained from the Lookup Service.

The Jini system does not require that there be a single Lookup Service. Many Lookup Services can co-exist on the same network, perhaps differentiated by the name of the group of which they are a part. Services can register in one, many, or all of the Lookup Services that they find, depending on the local policy of the service.

4 A Network Centric Computing Architecture

While each step from standard distributed computing to the Java environment as a platform for the World Wide Web to RMI as a general platform for distributed computing and to the Jini system as a way of organizing services and clients on a network seems obvious in retrospect, the impact of the changes that have gone on is fairly revolutionary. The kind of environment allowed by the Jini system is a radical break with the assumptions that have been held constant in computer architectures for

the past 50 years.

This standard computer architecture was based on the tight connection between a processor and a mass storage device. The software that would be run on the processor was assumed to be located on the mass storage device, and therefore that software could be specialized for the kind of processor on which it would run. Low level languages (such as C) could be translated by compilers into the appropriate set of instructions for a particular processor. Such binary code would be installed on the appropriate computer, and installing a program on a processor that was not the same as the one for which the program was compiled was a way of insuring that, at best, the program would not run.

This tight connection between the processor and the storage device from which the processor would obtain all of the code run on the processor even had its effect on what we took to be computers. A computer had to have both a processor and some form of storage for its programs. Other devices might be attached as well, but these were peripheral to the core function of the computer. This tie is apparent in the very boot sequence of our computers--after running a set of built-in diagnostics, the first thing a computer will do is look for its local disk, from whence it will load the programs that continue the boot process.

Defining a computer in this way actually excludes a large number of the devices that are currently being produced that contain microprocessors. These devices, from personal digital assistants and cellular phones to automobiles and kitchen appliances, often contain considerable computational power, often more than was available in personal computers of less than a decade ago. But they do not contain much mass storage. What little code is used to run these devices resides in a persistent store that is limited in size and expensive.

However, many of these devices are either capable of being connected to a network or are already connected to some network. A cellular phone is an indication of what, I believe, is soon to be the common case. When such a device boots, it runs through some basic diagnostics. But it then looks not for local mass storage, but for a network. It is from the network connection that the device is able to offer value and give service. But it can also be from the network that the device can locate the code that is needed to expand the services offered by the device, or offer code to other devices to allow those devices to make use of its service.

Using techniques similar to those in the Jini architecture, we can move code into these devices to allow them access to new services over the network, and move code out of these devices to allow them to offer services to others on the network. This in turn greatly expands the notion of what a network citizen can be. Rather than thinking of networks as the domain of the desktop system and the large server, providing services in business and engineering, the network becomes a substrate for electronic communities that span all of our different areas of life, connecting our homes, cars, offices, and selves to communications infrastructure of untold size and diversity.

This change could be a true paradigm shift, in the original sense described by Kuhn[9] as opposed to the more popular sense of „the next new thing.“ As a paradigm shift, it is a change the way we view the world, and the very questions that we ask about that world. Issues of privacy and security, ignored for so long because of their difficulty, cannot be ignored if the internet reaches into our home appliances and our methods of transportation. Issues of resource management and resource discovery will need to be addressed that can, and have, been ignored by desktop and server systems that can assume a large local disk and virtual memory. The questions of how one tests and verifies a system that receives its code as needed rather than through

explicit installation will need to be addressed.

These are not the questions that concern us now, since we have been living under a different paradigm. But the world is changing. We have been living in a world of machines that have formed stand-alone islands of functionality, perhaps being tied to a network for some communication to other machines. We are moving to a world in which the network is necessary for the functioning of most systems, where the stand-alone machine will be a minority member of the computing world. Code and data will move around the network, allowing us to update parts of the network in a dynamic fashion, with clients getting the updates transparently when they are needed.

We are only beginning to understand the scope of this new world, and are only starting to ask the questions that need to be answered to make this world a reality. There will still be much to do in the context of the old world, and work in that paradigm will continue for some time. But the new paradigm will, I believe, begin to drive both the market and the technology. The world of the new paradigm is more varied and rich than the world that we now inhabit, and the possibilities are both enticing and challenging. Our users (and the market) have shown us that the network world is where they wish to be. It is up to us to define that world, and make it as rich as possible.

References

1. Carroll, L., Through the Looking Glass, in The Complete Works of Lewis Carroll, The Modern Library, (1954)
2. Schroeder, M., A State-of-the-Art Distributed System: Computing with BOB, in Mullender, S. (ed.), Distributed Systems, Addison-Wesley (1993) 1-16
3. The Object Management Group, Common Object Request Broker: Architecture and Specification, OMG Document Number 91.12.1 (1991)
4. Rogerson, D., Inside COM, Microsoft Press (1997)
5. Gosling, J., Joy, B., and Steele, G., The Java Language Specification, Addison-Wesley (1996)
6. Lindholm, T. and Yellin, F., The Java Virtual Machine Specification, Addison-Wesley (1997)
7. Wollrath, A., Riggs, R., and Waldo, J., A Distributed Object Model for the Java System, Computing Systems, Volume 9 Number 4, (1996) 265-290
8. Sun Microsystems, Jini Technology Architecture Overview, http://www.sun.com/jini/whitepapers/architecture.html
9. Kuhn, T., The Structure of Scientific Revolutions, The University of Chicago Press, (1962)