Object Technology – A Grand Narrative?

Steve Cook

Microsoft UK Ltd, Cambridge
steve.cook@microsoft.com

Abstract. This brief article sets out some personal observations about the development of object technology from its emergence until today, and suggests how it will develop in the future.

1 The Beginning

Like many people, my first encounter with object oriented programming was in 1981 when I read the August 1981 special issue of Byte magazine [1] that was entirely devoted to Smalltalk (I still have it). At the time, I was developing graphical user interfaces using the programming languages C and Pascal, and I encountered difficulties in trying to make the procedural structure of my programs correspond cleanly to the problems I was trying to solve. It seemed that something was missing from these languages; and when I saw Smalltalk, I realized what that thing was – objects. In fact I discovered that I was late to the party; objects had been in the Simula language since 1967.

I threw myself for the next ten years into studying and teaching object-oriented programming. My team implemented Smalltalk and used it to build experimental distributed systems. I witnessed the development and competition of Objective-C and C++. Eiffel inspired me by its visionary design. I tried to understand the relationship between functional and OO programming. I participated in the first several OOPSLA and ECOOP conferences.

In those days, objects were controversial. Some academic colleagues of mine were deeply offended by the polymorphism: they saw it as fundamentally undermining the programmer's ability to reason logically about program consequences. Others were unwilling to admin that anything new was going on: objects were variously "just subroutines" or "just data abstractions" or "just entities".

Of course, objects won the day. Although COBOL remains popular for commercial data processing, most widespread programming languages today are object-oriented. All of the basic mechanisms of object-oriented programming are generally taken for granted: inheritance, virtual functions, instances and classes, interfaces, real-time garbage collection. In Microsoft I work daily with the .NET framework, an extensive object-oriented library for building distributed interactive applications, using the languages supported by Microsoft's Common Language Runtime, primarily C# and Visual Basic.

However we should recognize that the path that we've collectively trod over the past 20 years has involved some significant blind alleys, each of which has had a substantial effect on the industry. I've been personally involved in all of them.

D. Thomas (Ed.): ECOOP 2006, LNCS 4067, pp. 174 – 179, 2006.

2 Some Detours

People became very excited about objects. In some quarters, objects became regarded as a solution for all known problems. Let's look critically at some of the proposals that resulted.

2.1 Objects Solve the Reuse Problem

We've all been enticed by promises of reuse. Objects, we were told, are just like hardware components. We were led to believe that once an object has been designed to solve a problem, it can be reused over and over again in different projects, thus reducing the cost and increasing the productivity of software development.

Although there is a grain of truth in this proposition, it has often been treated much too naively. I have seen several large companies kick off major initiatives based on a simplistic interpretation of this suggestion, and spend many millions of dollars investing in structures and organizations for reuse that ultimately did not work. Objects are typically not reusable because of architectural mismatches. You can almost never take an object designed for use in a single application and use it successfully in another. Except for a few very small and self-contained items, almost all objects are constructed with a large number of both functional and non-functional assumptions about the environment in which they live. Such assumptions are encapsulated by object-oriented frameworks and class libraries, without which it would be almost impossible to build modern distributed interactive applications. The design of these frameworks and libraries is a very complicated business, and organizing them to be powerful and easy to use requires enormous skill and investment.

2.2 Objects Solve the Distribution Problem

Objects seemed like a great answer to the problem of how to build a distributed computing system. If we could conceal the difference between invoking an operation on an object locally and remotely, it was said, then distribution became simple. Indeed, there were early experimental implementations of distributed Smalltalk in which the menus would pop up on the wrong screens because of programming errors. The highest-profile manifestation of this principle was CORBA – the Common Object Request Broker Architecture, from the Object Management Group.

The facts about distribution are these:

- invoking a remote operation is orders of magnitude slower than invoking a local one;
- the programmer of a remote operation probably did it in a programming language different from yours;
- the invoker and the invokee will want to change their implementations at different times;
- neither end can trust the other nor the connection to be well-behaved in any way.

The consequence of these facts is that using objects and their interfaces as a basis for implementing distributed systems does not work well. Abstraction is a wonderful