2

Designing Distributed Objects

Learning Objectives

In this chapter, we will study the central notion of distributed objects. We will learn about the history of object orientation and how contributions from different disciplines of Computer Science led to distributed objects. We will study the notations defined in the Unified Modeling Language (UML), which we will use to communicate about the design of distributed objects. We will then grasp a meta-model for objects in order to acquaint ourselves with the concepts of distributed objects, such as object types, attributes, operations, inheritance and polymorphism. We will recognize the differences between distributed and local objects in order to understand how the design of objects differs when objects reside on different hosts.

Chapter Outline

In the last chapter, we introduced the concept of a distributed system. The main constituents of every distributed system are the system components. They offer services to other components, which can request service execution.

In this book, we assume an object-oriented view of distributed systems. Hence, we consider the components of a distributed system to be objects. The object-oriented paradigm is a very appropriate model for a distributed system. Services can be seen as operations that an object exports. How these objects are implemented is of no concern and the type of an object is therefore defined by interfaces. Interfaces may declare the visible state of a component, which can be regarded as a set of object attributes. The concept of references to objects is used for addressing components. Finally, the requesting of a service can be seen as a remote operation invocation.

All aspects of object orientation (concepts, languages, representations and products) have reached a stage of maturity where it is possible to address complex issues of distribution using the object-oriented paradigm. We review the evolution of object-oriented concepts in the next section to indicate exactly where we have now reached. Prior to implementation, all systems necessarily go through preliminary stages of more general analysis and design which, in the case of those that use an object-oriented approach, are now facilitated by the standardized representations offered by the Unified Modeling Language, in particular the diagrams defined in the UML Notation Guide. We discuss those diagrams in the second section. We then discuss a meta-model relevant to distributed objects in order to provide a framework for explaining basic concepts, as now generally understood, and a hierarchy for thinking about how objects are implemented. Once detailed design is underway it is necessary to consider very carefully the differences between centralized and distributed systems and the implications of factors including the life cycle, object references, request latency, security and so on. We discuss what differentiates distributed object design from local object design in the last section.

2.1 Evolution of Object Technology

Figure 2.1 shows an overview of important steps in the development of distributed objects. We can distinguish three strands in object orientation: distributed systems, programming languages and software engineering. Although largely independent, several cross-fertilizations are indicated by the lines that span the boundary of the strands.

Object orientation started in Scandinavia with the development of Simula [Dahl and Nygaard, 1966]. Simula is a programming language that was targeted to the development of simulation programs. Simula includes the concept of a class, which is a type whose instances can react to messages that are defined by operations.

David Parnas presented the idea of a *module* in his seminal paper 'A technique for the software module specification with examples' [Parnas, 1972]. The need for modules is driven by the observation that data structures change more frequently during program maintenance than names, parameters and results of operations. By hiding information about the data structures within a module, other modules cannot become dependent on these modules; they are not affected if data structures have to be changed. These ideas later

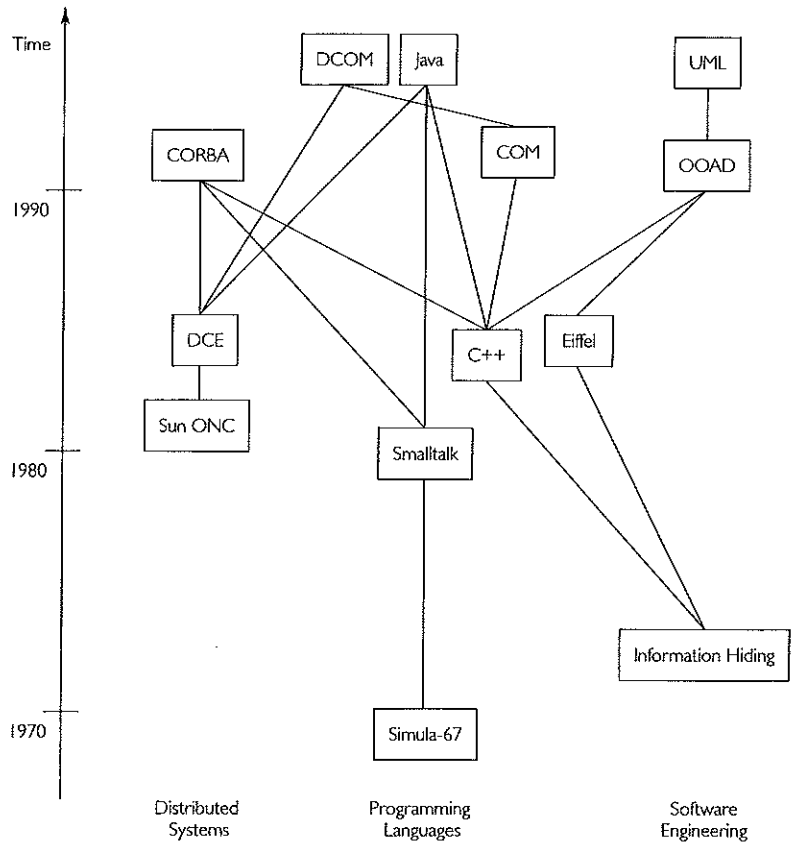

Figure 2.1
The Evolution of Object Technology

influenced programming language designers and led to the inclusion of constructs to restrict the visibility of instance variables and methods in object-oriented programming languages, such as C++, Eiffel and Java.

In the late 1970s, the Xerox Palo Alto Research Center conducted research into novel user interfaces for computers and invented the ideas of desktops, windows and pointing devices, such as mice or trackballs. It became clear that existing imperative programming languages were not well suited for applications that needed to react to events in multiple windows. In response to that problem Smalltalk [Goldberg, 1985] was developed as a new programming language that was centred around the notion of an *object*. Objects were applied to many different problems and Smalltalk implemented the vision that 'everything is an object'. Objects were used to implement the functionality of applications, to store data and to compose the user interface from different components. Objects were even used to manage the evolution of the set of object types that are defined. Smalltalk also included the concepts of inheritance and polymorphism that made it possible to create reusable class libraries, for instance to support user interface management.

Inspired by the success of Smalltalk, several groups introduced object-oriented concepts into procedure programming language, which led to the creation of hybrid object-oriented languages. Bjaerne Stroustrup led a development at AT&T Bell Labs, where his group

extended the C programming language with object-oriented concepts into C++ [Stroustrup, 1986]. Although Smalltalk was a conceptually nicer language, C++ was more successful commercially. C++ is a strict superset of C and promised a smoother migration to object-oriented concepts while continuing to use existing C code. C++ is more efficient than Smalltalk. Finally, C++ is a statically typed language, which means that static semantic errors can be detected at compile time, while they are only detected at run-time in Smalltalk.

For several reasons, C++ was, and to a considerable extent still is, rather difficult to use. The language leaves a considerable amount of freedom to programmers. Like Smalltalk, it does not have a rigorous type system, it enables programmers to manipulate object references and it lacks powerful memory management. Eiffel ([Meyer, 1988b]) was defined to overcome these deficiencies. Eiffel supports the notion of 'design by contract'. Contracts are specified in class definitions that are binding for client and server. The contract is enforced by Eiffel's powerful type system. Class definitions not only include static properties of classes but also behavioural definitions, such as invariants and pre- and post-conditions of operations.

The success of object-oriented programming languages motivated the development of notations and a method for applying the object-oriented paradigm in earlier phases of the development. At the end of the 1980s and early 1990s, object-oriented analysis and design (OOAD) methods were defined in order to enable the seamless use of objects from requirements to implementation. About ten methods were defined and the more successful ones included Booch's Object-Oriented Analysis and Design [Booch, 1991], Jacobson's Object-Oriented Software Engineering [Jacobson et al., 1992], Rumbaugh's Object Modeling Technique (OMT) [Rumbaugh et al., 1991] and the Open Method [Henderson-Sellers et al., 1998]. The techniques each used their own notation for objects and they had a different focus on analysis, design or both.

A consortium led by Rational Corporation, which employed Booch, Rumbaugh and Jacobson at the time, defined the Unified Modeling Language (UML) in an attempt to resolve the heterogeneity in object-oriented analysis and design notations. UML includes a meta-model, a semantics definition and a notation guide. UML supports modelling the structure of systems using class diagrams and object diagrams. It also facilitates modelling the behaviour of systems. Interaction diagrams are used for modelling inter-class behaviour. State diagrams are used to model intra-class behaviour. UML was adopted by the Object Management Group (OMG) as an industry standard [OMG, 1997c]. Methods now merely differ in how this standard notation is used during the analysis and design of objects.

Many distribution aspects of objects have their origin in the Remote Procedure Calls (RPCs) that were developed by Sun Microsystems as part of the Open Network Communication (ONC) architecture. In ONC, clients can call procedures that run on remote machines. The interfaces of remote programs are specified in a dedicated interface definition language. Client and server stubs convert complex procedure parameters and results to and from a form that is transmissible via network transport protocols, such as TCP or UDP. Stubs are derived by `rpcgen`, a compiler for the interface definition language.

Other Unix vendors and PC vendors quickly adopted RPCs as a primitive to build distributed systems at a higher level of abstraction than TCP. As part of their effort to standardize Unix, the Open Software Foundation (OSF) also standardized RPCs in the Distributed Computing Environment (DCE). In particular, DCE standardized an external data representation (XDR) for data types that are used in procedure parameters and results. By means of XDR, clients can call remote procedures from servers that have a different data representation. OSF also standardized a number of higher-level services, such as a naming service to locate components and a security service that can be used to authenticate components.

The OMG was created as early as 1989 in order to develop and promote the Common Object Request Broker Architecture (CORBA). CORBA combines the idea of remote invocation with the object-oriented paradigm. An essential part of the CORBA standard is an Object Request Broker (ORB). Objects use ORBs to request execution of operations from other objects. Hence, ORBs provide the basic communication mechanism between distributed objects. These objects may be heterogeneous in various dimensions: the deployment platforms, the operating systems, the networks by which they are connected and the programming languages in which they are written. Hence, ORBs provide the basic communication mechanisms for distributed and heterogeneous objects. The OMG has also standardized interfaces for higher-level services and facilities that utilize the basic communication mechanism to solve standard tasks in distributed systems, many of which support achieving the distribution transparency dimensions identified in the previous chapter.

Microsoft defined its Component Object Model (COM) as an interoperability model for applications that are written in different programming languages. COM objects can be implemented in different programming languages, including Visual Basic, Visual C++ and Visual J++. COM is not fully object-oriented, as it does not include multiple inheritance. COM objects, however, can have multiple different interfaces and this is used as a replacement of inheritance.

The first versions of COM did not support distribution. They served as a basis for integrating heterogeneous objects on the same machine. With the advent of Windows NT4.0, a technology called Distributed Component Object Model was introduced to COM and COM now uses OSF/RPCs to support distributed access to COM objects.

Distribution primitives are tightly integrated with object-oriented programming language features in Java [Winder and Roberts, 1997]. Java includes primitives for Remote Method Invocation (RMI) so that a Java object on one machine can invoke methods from a Java object that resides on a remote machine. RMI also uses stubs and skeletons to perform marshalling, although data representation heterogeneity does not have to be resolved.

We have now very briefly sketched the evolution of object technology. Several of the concepts that we could only touch on here will have to be revisited throughout the rest of the book. We will discuss the principles that underlie Java/RMI, OMG/CORBA and COM in Part II. The use of these middleware systems requires detailed knowledge of the underlying object-oriented concepts as well as object-oriented design notations. We will focus on them in the remainder of this chapter.