If we need to make changes in such fine-grained methods (for example, we might need to add a new parameter based on some new requirement), then all of our client programs that are using our API (also known as consumers of the API) will break down. These programs will need to change their code to pass in the modified method arguments so that our API works from within their code. This introduces tighter coupling between systems even in an n-tier architecture where the code might be used by other external programs.

To introduce loose coupling in such systems, we may sometimes use a coarse-grained model, which is described next.

Coarse-Grained Model

A coarse-grained model has fewer details , and in the context of an object-oriented system, coarse-grained means less classes and a limited number of methods, or methods with few parameters. For example, a coarse-grained version of the example class method discussed above, namely:

```
myAPIMethod.FindCustomers(int ProjectID, int pageNumber,
                          int pageLength, ref noOfPages);
```

will look like:

```
myAPIMethod.FindCustomers(int ProjectID);
```

As you can see, we removed the "fineness" of this method by removing the extra arguments, which were acting like "filters" on the data returned. So instead of getting only the specified number of paged results, we will get all the records of customers for a particular project ID. This means that the entire set is returned without any filter; hence it is a coarse-grained method when compared to the earlier method of passing paging-related parameters. We can make it even more coarse-grained by removing the project ID as the argument and returning all customers irrespective of the project to which they belong. The degree of coarseness or "fineness" depends on our actual project requirements.

So how is the granularity of the system related to flexibility and better change management? We will see this in the next section.

What is SOA?

SOA, or Service Oriented Architecture, is an architectural approach aimed at making it easy to introduce new features into an existing system, share data with other applications for easier integration, and bring a faster **Return On Investments** (**ROI**) to the business processes.

An important point to consider is that SOA does not formalize any approach or specific implementation technique. We can implement an SOA-based architecture using a number of different methods and approaches. SOA is an architectural guideline on how to achieve interoperability and cross-process integration in a scalable way. Most developers assume that SOA is equivalent to web services, but the fact is that web services are merely one of the convenient ways to implement an SOA-based architecture, due to the interoperability of XML across varied systems. But this does not imply that we cannot implement SOA without web services! If we don't have to communicate within a heterogeneous environment (having multiple platforms, software, and so on running across networks), then we can implement an SOA-like architecture using message-based systems and remoting. In a homogenous environment, web services may not be needed. But web services are the best options to use in an SOA because most enterprise systems that need to use SOA are operating in a heterogeneous environment.

A homogenous environment is the one where we have the same set of development platforms and operating system in all of the networked machines. For example, all machines within a company's internal LAN can have .NET runtime loaded with the Windows 2003 server. A heterogeneous environment is one where we have multiple Operating Systems running on different machines in the network. For example, we may have Windows on one, Linux on others, while some machines may have .NET runtime, and yet others may be hosting J2EE applications.

In this chapter, we will learn about SOA, and how it can help us achieve rapid application integration, code re-use, and automate business processes with the help of a practical example.

Why SOA?

We have been following OOAD, Object-Oriented Analysis and Design in the previous chapters, creating a domain model and using objects as the core framework for our applications. SOA does not make OOAD redundant; in fact, it complements it. OOAD is more of an implementation approach, whereas SOA is more of a high-level architectural approach.

In the previous chapters, we noticed that we used n-tier architecture to make our applications scalable, robust, and flexible. If n-tier architecture helps us build all of that, then why do we need SOA? The answer is: better change management!