- **Limited Future growth:** The Applications that don't need to grow with time, for example, a limited family album website, a small sports website, or an intra-company blog system.
- **Budget limitations:** Finally, one of the most important reasons for letting an application use a limited architecture and not increase its scope is money. A tiered scalable architecture is more complex, requires skilled programmers and takes more development time and hence more money. Often, project stakeholders (or clients) will not let an application use scalable architecture, as they want a quick turn-around time and less financial constraints. Hence, an application that might be categorized "big" according to the above criteria, would be limited in scope due to a limited budget.

Then there are big enterprise-level applications, which have wide scope across many verticals, and websites that may need to scale up in the future. Such big applications may depend on other third-party applications for updating, deleting or sharing data within the company. or even outside of it. In enterprise-level firms (big companies, such as those in the Fortune 1000), there are typically big software applications that collaborate with other applications, for example, accounting software may be integrated with **Enterprise Resource Planning** (**ERP**) software, or the Inventory system may be talking and sharing data with the financial software.

In such big systems, there can be a lot of inter-application communication, and because the number of users accessing these applications can grow with time, scalability, interoperability, and flexibility have to be built in. A small change in such applications can create a ripple effect and cause big changes in related systems.

So applications which are "small" in size and have a limited scope might get away with regular changes compared to large applications with complex business logic. Therefore, incorporating changes is quick and less costly. But in a large system, one change can break other applications that may be related to the system being changed. And the cost of making such changes is much more than in smaller systems, due to the complexity of the code base, and the possible impact on other applications within the same environment. Let us understand why the code base of such applications can get really complex making it difficult to incorporate future changes or modifications.

Tight and Fine-Grained Domain Model

Enterprise-level systems may have a very complex domain model, which contains lots of business classes and methods developed in an API-like fashion. In Chapter 4, we learnt that we could make our code base more flexible and adaptable to change by using an n-tier architecture with a proper domain model. But as our application grows in scope, this domain model itself might get very complex with deeply nested/related classes, thousands of methods and overloaded signatures, multiple parameters, and so on.

One way of measuring the complexity of an application is to understand the granularity of the application. Granularity defines the size of an application as well as the number of individual components in the system. For object-oriented software systems, we can use class methods as units of granularity. The more methods a class has, the more granular or fine-grained it is. Similarly, we can use the class itself as a measure of granularity. The more classes we have, the more fine-grained our system is. If it has less methods, then it is relatively coarse-grained (you can think of it in terms of currency; a $10 dollar note is coarse grained, and ten $1 notes are more fine-grained).

As we go along adding more functionality and classes to the system, the domain model may inflate to big proportions and become tightly intertwined, making it complex for use by external systems, and increasing the granularity of the system. Let us understand how.

Each class in such a business layer will have its own large set of methods with specific signatures, and in order to call these methods from outside the system (using an API), we need to provide all of the parameters in the method signature. For example, we can call a method to get a list of customers for a particular project, as in:

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

Let's assume that the above code returns a paged list of all customers for a particular project ID, where `pageNumber` and `pageLength` denotes which page and how many records we want to return (as we want to show paged records). This method call is fine-grained because we are specifying each and every parameter to the business method call, which in turn passes these parameters to a data access layer (DAL) method and returns the results. We use the word "fine" here to denote the "exactness" of the method call, and if we leave out even one parameter, the method call will result in an error.