Scalability denotes the ability to accommodate a growing load in the future.

Software architectures should be designed in such a way that they are stable for the lifetime of the system. The architecture therefore not only has to be able to bear the load when the system goes into production, but a prediction has to be made about how the load will develop during the lifetime of the system. Predictions as to how the load is going to develop are rather difficult. When the Internet was developed, no one estimated the current number of nodes. One of the reasons for the Internet's success is that it can grow to accommodate the increased load; it has a scalable architecture. In general, we refer to system architectures as *scalable* if they can accommodate any growth in future load, be it expected or not.

Distributed system architectures achieve scalability through employing more than one host.

Scalability requirements often lead to the adoption of distributed system architectures. In the Boeing case study, the old mainframe just could not bear the load imposed by an additional 1.5 billion parts each year. It created a bottleneck and had a negative impact on the overall performance of the engineering division. Distributed systems can be scalable because additional computers can be added in order to host additional components. In the Boeing example, 20 Sequent machines share the load of the databases needed to accommodate the overall engineering data. More machines can be added if it turns out to be necessary.

1.3.2 Openness

Open systems can easily be extended and modified.

A non-functional property that is often demanded from a software and system architecture is that it be open. *Openness* means that the system can be easily extended and modified. To facilitate openness, the system components need to have well-defined and well-documented interfaces. The system construction needs to adhere to recognized standards so that system components can be exchanged and the system does not become dependent on a particular vendor. Openness is demanded because the overall architecture needs to be stable even in the presence of changing functional requirements. Institutions that procure a new system want the system to evolve with the institution. They want to preserve their investment rather than throw it away. Hence, it is required that new components can be integrated into the system in order to meet new functional requirements.

The integration of new components means that they have to be able to communicate with some of the components that already exist in the system. In order to be able to integrate components a posteriori, existing components must have well-defined *interfaces*. These interfaces must declare the services that a component offers. A *service* is an operation that a component performs on behalf of a user or another component. Services are often parameterized and the service parameters need to be specified in the interface, too. A *client component* uses such an interface to *request* a service from a *server component*. Note that a component that is a client of some component can itself be a server to other components. Components might be reactive and have to respond to the occurrence of *events* that are detected by other components. A billing component in the video-on-demand server, for instance might have to react to the downloading of a video that is detected by the video database component. In an open system, components need to declare which events they produce so that other components can react to them.

Openness and distribution are related. Components in a distributed system have to declare the services they offer so that other components can use these services. In a centralized system, components may also declare their interfaces so that other systems can use them. The use, however, is often confined to procedure calls. These restrict the requesting component in various ways. A request in a distributed system is more flexible. It can be made from the same or a different machine. It can be executed synchronously or asynchronously. Finally, the requester and the server can be constructed in a heterogeneous way.

Distributed system components achieve openness by communicating using well-defined interfaces.

1.3.3 Heterogeneity

A requirement that occurs frequently for new systems is that they have to integrate heterogeneous components. Heterogeneity arises for a number of reasons. Often components are bought off-the-shelf. Components that are constructed anew might have to interface with legacy components that have been around in an enterprise for a long time. Components might be built by different contractors. In any of these cases, components are likely to be heterogeneous.

Component *heterogeneity* arises from the use of different technology for the implementation of services, for the management of data and for the execution of components on hardware platforms. Heterogeneity can stem from the autonomy of the component builders, from components built at different points of time or from the fact that one technology is better suited for a component than another technology.

Heterogeneity of components can arise in the programming languages, operating systems, hardware platforms and network protocols.

When we revisit the case studies that we have discussed in Section 1.2 we see that most of them involved building distributed systems from heterogeneous components. In the Boeing case, many components were bought off-the-shelf. They were constructed using different programming languages and operated on different hardware platforms. In the bank example, new components had to be integrated with legacy systems for account management that operated on a mainframe. In the video-on-demand system, front-end components were written in Java so as to be portable. Back-end components had to be written in C++ to be high-performant.

The integration of heterogeneous components, however, implies the construction of distributed systems. If the system includes components that need to be executed on their native hardware platforms, the components needs to remain there. To be able to appear as an integrated whole to users, components then have to communicate via the network and heterogeneity has to be resolved during that communication.

Component heterogeneity can be accommodated by distributed systems.

1.3.4 Resource Access and Sharing

Resources denote hardware, software and data. *Resource sharing* is often required in order to render an expensive resource more cost-effective. A Sequent database machine is a highly specialized and, therefore, rather expensive piece of equipment. It becomes cost-effective for Boeing because it is shared by several hundred engineers. Resource sharing also occurs as a consequence of communication and co-operation between users. In the Boeing case study, the construction engineer of the aircraft communicates with the maintenance engineer by means of the database component. It stores the engineering documents that

Often resources, i.e. hardware, software and data, need to be shared by more than one user.