CHAPTER ONE

Introduction

This chapter defines the term software component and summarizes the key arguments in favor of component software. Components are well established in all other engineering disciplines, but, until the 1990s, were unsuccessful in the world of software. The reasons behind this failure can be linked to the particular nature of software. The chapter concludes with a discussion of the nature of software, its consequences for component software, and lessons learned from successful and unsuccessful approaches.

1.1 Components are for composition

One thing can be stated with certainty: components are for composition. *Nomen est omen*. (Literally, "the name is a sign"; usually interpreted as: one's name predicts one's fate.) Composition enables prefabricated "things" to be reused by rearranging them in ever-new composites. Beyond this trivial observation, much is unclear. Are most current software abstractions not designed for composition as well? What about reusable parts of designs or architectures? Is reuse not the driving factor behind most of these compositional abstractions?

Reuse is a very broad term covering the general concept of a reusable asset. Such assets can be arbitrary descriptions capturing the results of a design effort. Descriptions themselves normally depend on other, more detailed and more specialized descriptions. To become a reusable asset, it is not enough to start with a monolithic design of a complete solution and then partition it into fragments. The likely benefits of doing so are minimal. Instead, descriptions have to be carefully generalized to allow for reuse in a sufficient number of different contexts. Overgeneralization has to be avoided to keep the descriptions nimble and lightweight enough for actual reuse to remain practicable. Descriptions in this sense are sometimes called components (Sametinger, 1997).

This book is not about reuse in general, but about the use of software components. To be specific, for the purposes of this book, software components are executable units of independent production, acquisition, and deployment that can be composed into a functioning system. To enable composition, a

software component adheres to a particular component model and targets a particular component platform. (The details will be explored later; also, there are several other attempts at defining this and related concepts; see Chapter 11.) Composite systems composed of software components are called component software. The requirement for independence and executable form rules out many software abstractions, such as type declarations, C macros, C++ templates, or Smalltalk blocks. Other abstractions, such as procedures, classes, modules, or even entire applications, could form components, as long as they are in an executable form that remains composable. Indeed, procedural libraries are the oldest example of software components. Insisting on potential independence and executable form is essential in order to allow for multiple independent vendors, independent development, and robust integration. These issues are therefore covered in great detail in this book.

What is the motive for producing, distributing, buying, or using software components? What are the benefits of component software? The simplest answer is that components are the way to go because all other engineering disciplines introduced components as they became mature – and still use them. Shortly after the term software crisis was coined, the solution to the often-cited crisis was also envisioned: software integrated circuits (ICs) (McIlroy, 1968; Cox, 1990)! Since then, for 30 years, people have wondered why this intuitive idea never truly came to fruition.

1.2 Components – custom-made versus standard software

In the following discussions it is assumed that component software technology is available. The question addressed in this section is "What are the benefits of using components?"

Traditional software development can broadly be divided into two camps. At one extreme, a project is developed entirely from scratch, with the help of only programming tools and libraries. At the other extreme, everything is "outsourced" – in other words, standard software is bought and parametrized to provide a solution that is "close enough" to what is needed. Full custom-made software has a significant advantage (when it works): it can be optimally adapted to the user's business model and can take advantage of any in-house proprietary knowledge or practices. Hence, custom-made software can provide the competitive edge in the information age – if it works.

Custom-made software also has severe disadvantages, even if it does work. Production from scratch is a very expensive undertaking. Suboptimal solutions in all but the local areas of expertise are likely. Maintenance and "chasing" of the state-of-the-art, such as incorporating web access, can become a major burden. Interoperability requirements further the burden, with other in-house systems and, more critically, also with business partners and customers. As a result, most large projects fail partially or completely, leading to a substantial risk. Also, in a world of rapidly changing business requirements, custom-made software is often too late – too late to be productive before becoming obsolete.

With all these guaranteed disadvantages in mind, which are offset by only potential advantages, the major trend toward "outsourcing" in the industry is understandable. Production of custom-made software is outsourced under fixed-price contracts to limit the financial risk. To cover the time-to-market risk, there is a strong trend toward using standard software – that is, software that is only slightly adjusted to actual needs. The burden of maintenance, product evolution, and interoperability is left to the vendor of the standard package. What remains is to carry over parametrization and configuration detail when moving to the next release – still a substantial effort, but unavoidable in a world of change.

What, then, is wrong with standard software? Several things. First, standard software may necessitate a greater or lesser reorganization of the business processes affected. Although business process re-engineering can be a very worthwhile undertaking, it should be done for its own sake rather than to make the best of suboptimally fitting standard software. Second, standard software is a standard: competitors have it as well and no competitive edge can possibly be achieved by using it (except by using it extraordinarily well). In any case, this is acceptable only when tight regulations eliminate competitive advantages. Third, as standard software is not under local control, it is not nimble enough to adapt quickly to changing needs.

Here is an example of standard software forcing its footprint on to a large and well-established organization. In 1996, Australia Post decided to use SAP's R/3 integrated solution. With R/3, Australia Post can keep track of each individual transaction, down to the sale of a single stamp. Australia Post is a large organization with a federated structure; each Australian state has its own head office reporting to the central head office.

Traditionally, state head offices reported on the basis of summaries and accounts "in-the-large." For example, detailed sales figures for each branch office were not passed on beyond the state head office. R/3, however, supports only a monotonic hierarchy of access authorizations. It is not possible to grant the national head office access to the accounts in-the-large without also granting access to every individual transaction. This was disturbing news for state head offices as their tradition of relative autonomy in making local decisions was undermined. Indeed, the strictly hierarchical business model enforced by R/3 clashed with the concept of a federated organization that delegates much responsibility and authority to its members – in this case, the state posts. SAP's comment, when asked if this aspect of R/3 could be changed, was: "Our systems implement best practice – why would you want to deviate from that?" The key point is that adoption of a standard solution may force drastic changes on the culture and operation of an organization.

With only two poles available, custom-made software loses out to a great extent. Standard packages create a level playing field and a necessary competitive edge has to come from other areas. Increasingly, software services are seen as something that is necessary simply to survive. Clearly, this is far from ideal when information and information processing have a great effect on most businesses and even define many of the newer ones.

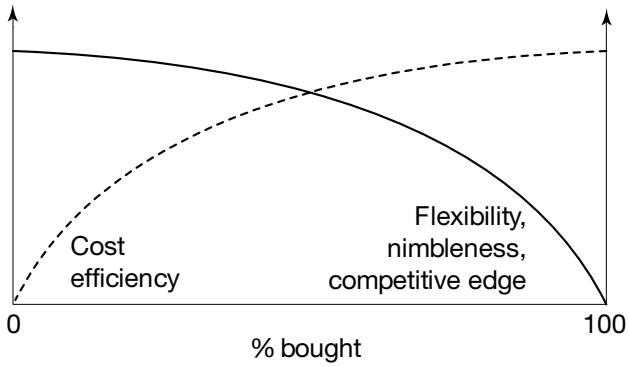

Figure 1.1 Spectrum between make-all and buy-all.

The concept of component software represents a middle path that could solve this problem. Although each bought component is a standardized product, with all the attached advantages, the process of component assembly allows the opportunity for significant customization. It is likely that components of different qualities (level of performance, resource efficiency, robustness, degree of certification, and so on) will be available at different prices. It is thus possible to set individual priorities when assembling based on a fixed budget. In addition, some individual components can be custom-made to suit specific requirements or to foster strategic advantages. Figure 1.1 illustrates some of the tradeoffs brought about by the spectrum of possibilities opened up by component software.

The figure is in no way quantitative, and the actual shape of the two curves is somewhat arbitrary. Intuitively, however, it is clear that non-linear effects will be observed when approaching the extremes. For example, at the left end of the scale, when everything is custom-made, flexibility has no inherent limits, but cost efficiency plummets.

Component software also puts an end to the age-old problem of massive upgrade cycles. Traditional fully integrated solutions required periodic upgrading. Usually this was a painful process of migrating old databases, ensuring upwards compatibility, retraining staff, buying more powerful hardware, and so on. In a component-based solution, evolution replaces revolution, and individual upgrading of components as needed and "out of phase" can allow for much smoother operations. Obviously, this requires a different way of managing services, but the potential gains are immense.

1.3 Inevitability of components

Developing excellent component technology does not suffice to establish a market. The discipline is full of examples of technically superior products that failed to capture sufficiently large markets. Besides technical superiority, a component approach needs critical mass to take off. A component approach gains

critical mass if the offered components are of sufficient variety and quality, there is an obvious benefit of using the components, and the offering is backed by sufficiently strong sources or sufficiently many second sources. Once critical mass is reached in a market segment, use of components in that segment quickly becomes inevitable. A "vortex" forms that pulls in traditional solutions in the area.

Not using available components requires reinvention of solutions. This can only be justified when the made solution is greatly superior to the buyable alternatives. Also, in a competitive market, components will improve in quality much faster than "hand-crafted" solutions. The result is the above-mentioned vortex: it becomes increasingly difficult to escape from using components.

As long as all solutions to problems are created from scratch, growth can be at most linear. As components act as multipliers in a market, growth can become exponential. In other words, a product that utilizes components benefits from the combined productivity and innovation of all component vendors. The component vendors are focused, supply many different customers, and are thus able to perfect their components rapidly. Therefore, even where an organization manages to sustain its proprietary technology, its relative market share will quickly dwindle in a market rapidly dominated by component technology. Avoiding the proximity of a component vortex promises calm waters but also eliminates the impulse that can be gained from the mighty pull of the vortex.

Preparedness for an emerging component market can be the deciding success factor for a company approaching such a vortex. Insistence on proprietary approaches can be catastrophic. Part of being prepared is the adoption of software engineering approaches that are component friendly – that is, they support modularity of requirements, architectures, designs, and implementations. Preparing for components thus leads to substantial advantages as a result of a better software engineering process, even if component markets are still seen as beyond the "planning horizon."

Out of preparedness a more proactive role can be developed. The first organization to create a convincing set of components for a certain market segment can set standards and shape the then emerging market to its own advantage. Instead of waiting for others or claiming that it is unlikely that, in a particular domain, a component market will ever form, stronger organizations may want to take the lead. An interesting example was the move by Sun to make its Solaris operating system "modular" (Wirthman, 1997). Instead of offering a collection of specialized operating systems, Sun factored Solaris into modules that can be combined according to needs (Mauro and McDougall, 2001). In a similar fashion, Microsoft factored Windows CE and provides means to custom-assemble a Windows CE version for a particular device to trim resource consumption and match device capabilities and purpose (Boling, 2001; Wilson and Havewala, 2001). These are first steps. If Sun or Microsoft allowed third-party modules (beyond device drivers), then this could well create a component market supporting the creation of highly customized operating infrastructure for specialized devices and appliances.

1.4 The nature of software and deployable entities

Software components were initially considered to be analogous to hardware components in general and to integrated circuits in particular. Thus, the term software IC became fashionable. Other related notions followed, such as software bus and software backplane (Figure 1.2).

Also popular is the analogy between software components and components of stereo equipment. More far fetched are analogies with the fields of mechanical and civil engineering – gith gears, nuts, and bolts, for example. However, comparisons did not stop at engineering disciplines and continued on into areas as extreme as the world of toys. The Lego block model of object technology was conceived but has also been strongly criticized. These analogies helped to sell the idea of software components by referring to other disciplines and areas in which component technology has been in use for some time and had begun to fulfill its promises.

All the analogies tend to give the impression that the whole world, with the one exception of software technology, is already component oriented! Thus, it ought to be possible – if not straightforward – to follow the analogies and introduce components to software as well. This did not happen for most of the industry, and for good reason. None of the analogies aids understanding of the true nature of software.

Software is different from products in all other engineering disciplines (for a comprehensive analysis, see Messerschmitt and Szyperski, 2002). Rather than delivering a final product, delivery of software means delivering the blueprints for products. Computers can be seen as fully automatic factories that accept such blueprints and instantiate them. Special measures must be taken to prevent repeated instantiation – the normal case is that a computer can instantiate delivered software as often as required. The term software IC and the associated analogy thus fail to capture one of the most distinctive aspects of software as a metaproduct. It is important to remember that it is these metaproducts that are actually deployed when acquiring software. The same holds true for software components.

It is as important to distinguish between software and its instances as it is to distinguish between blueprints and products, between plans and a building, or between beings and their genes (between phenotypes and genotypes). Whereas such lines are clearly drawn in other engineering disciplines, software seems "soft" enough to tolerate a confusion of these matters.

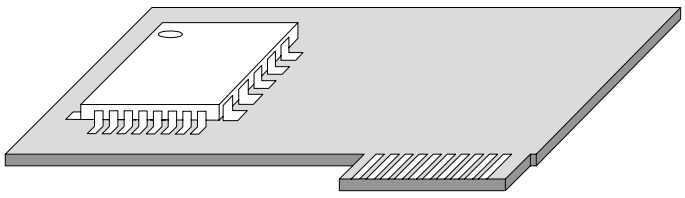

Figure 1.2 The software IC connected to a software bus.

There has been confusion about abstractions and instances since entity–relationship modeling (this was pointed out to the author by Alan Wills). To reintroduce a distinction that should have been in place from the beginning, phrases such as "entity occurrence" and "entity definition" are used. This confusion is even encouraged in the world of object technology. The corresponding distinction between class and object is frequently omitted, although there is occasional clarification of something as an "object instance" or an "object class." The established practice of not distinguishing between objects and classes leads the way, so the large number of nebulous publications on objects is not astonishing. To take an arbitrary example, consider the following astounding quotation (Cheung, 1996, p. 72):

> "The port class has 1024 virtual-circuit classes."

The article refers to an object model diagram as defined by the object modeling technique (OMT) (Rumbaugh *et al.*, 1991). A small excerpt of the diagram is shown in Figure 1.3.

What the author meant was: "The port object has 1024 virtual-circuit objects." There is nothing wrong with the cited article. The most likely explanation is that this "glitch" was the result of an attempt by the editor to introduce sharpness of terms and not call everything an object. This sort of mistake is easy to make – OMT object model diagrams describe the static relations of classes, but, when annotations refer to numbers of partners in a relation class, instances (objects) are meant. The UML diagrams and notation used throughout this book are much clearer in that they advocate a clear distinction between objects and classes.

The confusion between objects and classes is closely related to the nature of software. For example, both the plan of a building and the building itself can be modeled as objects. At the same time, the plan is the "class" of the building. There is nothing wrong with this, as long as the two kinds of objects are kept apart. In the world of logic, but also in database theory, this is called stratification – that is, introduction and maintenance of strata or levels of organization. Construction (and breach) of such layers has to be based on deep understanding. Some might argue that this lighthearted way of dealing

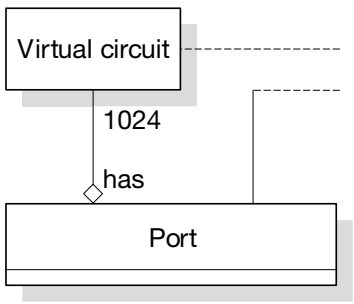

Figure 1.3 OMT object model with quantified "has" relationship. (After Rumbaugh, J., Blaha, M., Lorenson, W., Eddy, F., and Premerlani, W. (1991) *Object-Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, NJ.)

with object-oriented (OO) terminology had to be expected. After all, "object" is one of the most indefinite and imprecise terms that people could possibly use to name a concept. It seems fairly objective to say that.

To understand why it is so important to differentiate between plans and instances, it is useful to take a brief look at some of the ramifications of delivering plans rather than instances. Plans can be parametrized, applied recursively, scaled, and instantiated any number of times. None of this is possible with actual instances. As, with software, it is the plans that are delivered, instances can be of different shapes by using different parametrizations of the plan. In other words, software is a generic metaproduct that can be used to create entire families of instances.

If analogies to components in other engineering disciplines break down, then what about mathematics? Although defended by some, the purely mathematical approach fails exactly where the engineering analogies help – and vice versa. Mathematics and logic draw their strength from the isolation of aspects, their orthogonal treatment, and their static capturing. These are excellent tools to understand the software concepts of uniformity of resources, arbitrary copying, recursive nesting, parametrization, or configuration. However, mathematical modeling fails to capture the engineering and market aspects of component technology – that is, the need to combine all facets, functional and extra-functional, into one interacting whole, forming a viable product.

In conclusion, software technology is an engineering discipline in its own right, with its own principles and laws. Analogies with other engineering disciplines help us to understand certain requirements, such as those of proper interaction with markets and consideration of complex feature interactions. At the same time, these analogies break down quickly when going into technical detail. Deriving software architecture (component-oriented or not) by analogy with approaches in other disciplines is downright dangerous. The distinguishing properties of software are of a mathematical rather than a physical nature. However, placing emphasis solely on the mathematical underpinning is never enough to carry any engineering discipline. In particular, from a purely formal point of view, there is nothing that could be done with components that could not be done without them. The differences are concepts such as reuse, time to market, quality, and viability. All of these are of a non-mathematical nature – and value. Mathematics is not goal-driven, whereas engineering is: the goal is to create products.

1.5 Components are units of deployment

A software component is what is actually deployed – as an isolatable part of a system – in a component-based approach. Contrary to frequent claims, objects are almost never sold, bought, or deployed. The unit of deployment is something rather more static, such as a class, or, more likely, a set or framework of classes, compiled and linked into some package. Objects that logically form

parts of component "instances" are instantiated as needed, based on the classes that have been deployed with a component. Although a component can be a single class, it is more likely to be a collection of classes, sometimes called a module. (Components have further characteristics that distinguish them from modules – details can be found in section 20.3.) Components as a whole are thus not normally instantiated. Also, a component could just as well use some totally different implementation technology, such as pure functions or assembly language, and look not at all object-oriented from the inside (Udell, 1994):

"Object orientation has failed but component software is succeeding."

If classes are so similar to components, why did object technology not succeed in establishing significant component markets? The answers are manifold. First, the definition of objects is purely technical – briefly, encapsulation of state and behavior, polymorphism, and inheritance. The definition does not include notions of independence or late composition. Although such conditions can be added (Chapter 4, section 4.1), their lack has led to the current situation in which object technology is mostly used to construct monolithic applications.

Second, object technology tends largely to ignore the aspects of economies and markets and their technical consequences. Early proponents of object orientation predicted object markets, places that would offer catalogs full of objects or, more likely, classes, class libraries, and frameworks (Cox, 1990). The opposite has in fact occurred (Nierstrasz, 1991). Today we have only a small number of such sources. Most of them are driven by vendors that provide such semifinished software products to sell something else. The classic Microsoft Foundation Classes (MFC) is a good example. MFC primarily serves as a vehicle to simplify and unify programming for Microsoft's operating systems, component model, and application environments. There is no doubt that the vision of object markets did not happen. On the contrary, most of the few early component success stories were not even object-oriented, although some are object-based. (The relatively successful class libraries and frameworks are not software components in the strict sense used here.)

More recently, and based on true component technologies, successful markets began forming. Companies such as ComponentSource.com or Flashline.com sell thousands of ready-made components, mostly in the COM and Java categories, but VCL (a Delphi/C++ Builder technology by Borland) and .NET components are also present. CORBA components, however, are essentially non-existent on these markets. Companies such as ILOG and Rogue Wave Software generate substantial revenues by focusing on the production of components. ILOG approached $80 million in revenues in its fiscal year ending mid 2001. ILOG focuses on C, C++, and Java components for simulation/optimization, visual presentation, and business rule-based applications.

To some extent, the misprediction of object markets is understandable. For a technologist, markets are too easily considered marginal, as something left

for others to worry about once the technological problems have been solved. However, components are as much about the potential of technology as they are about technology. The additional investment required to produce components – rather than fully specialized solutions – can only be justified if the return on investment follows.

Typically, a component has to have a sufficiently large number of uses, and therefore clients, for it to be viable. As a rule of thumb, most components need to be used three times before breaking even. That is, two separate, from-scratch development efforts are still cheaper than a single effort to produce a more generic component. Repeated use is the central idea behind the notion of "reuse." For clients to use a component instead of a specialized solution, the component needs to have substantial advantages. One advantage could be technological superiority, but other advantages are more likely to help, such as the first solution to a known open problem, broad support base, brand name, and so on. Obviously, for larger organizations, "markets" could be found in-house – interestingly, most large organizations are now organized into cost centers and selling to internal clients is not much simpler than selling to external clients.

1.6 Lessons learned

Where are the mentioned component success stories? For many years, the most popular has been Microsoft's Visual Basic. Later, successes based on Java, Enterprise JavaBeans (EJB) and COM+ have followed. However, the oldest success stories are all modern operating systems. Applications are coarse-grained components executing in the environment provided by an operating system. Interoperability between such components is as old as the sharing of file systems and common file formats, or the use of pipe-and-filter composition. Other older component examples are relational database engines and transaction-processing monitors. Further, more recent successes, using finer-grained components, are plugin architectures. These have been in widespread use since the introduction of Netscape's Navigator web browsers. One of the first successful plugin architectures was Apple's QuickTime. Plugins (under the name of "extensions") have also been cultivated in the Mac OS, where they originated from "inits," patches to the system software in ROM that are loaded at boot time. DOS terminate-and-stay-resident applications (TSRs) were of comparable nature. Active Server Pages (ASP) and Java Server Pages (JSP) architectures for web servers follow a similar approach, accepting application-specific plugins into the server to provide server-side computations and web page synthesis to service incoming web requests. Finally, modern application and integration servers around J2EE and COM+ / .NET offer refined component models that bring much-needed discipline and opportunities for component use to the complex realm of enterprise applications.

What do all the above examples have in common? In all cases there is an infrastructure providing rich foundational functionality for the addressed

domain. Components can be purchased from independent providers and deployed by clients. The components provide services that are substantial enough to make duplication of their development too difficult or not cost-effective. Multiple components from different sources can coexist in the same installation. None of the named systems really shines when it comes to arbitrary combinations of components. In all cases, such combinations can lead to misbehavior. Apparently, for a working component market, it is sufficient that composability is highly likely rather than absolutely guaranteed.

Besides all this, there is another aspect that is often overlooked. In all the successful examples, components exist on a level of abstraction where they directly mean something to the deploying client. With Visual Basic, this is obvious – a control has a direct visual representation, displayable and editable properties, and has meaning that is closely attached to its appearance. With plugins, the client gains some explicable, high-level feature and the plugin itself is a user-installed and configured component.

Most objects have no meaning to clients who are not programmers. Class libraries and frameworks are typical developer tools and require highly trained and qualified programmers for their proper use. It is appropriate that component construction is left to persons of such standing. However, for components to be successful, composition and integration – that is, component assembly must not generally be confined to such a relatively small elite group. Today, there are many more authors of scripts than there are programmers. These customers are more interested in products that are obviously useful, easy to use, and can be safely mixed and matched – they are not in the least interested in whether or not the products are internally object-oriented.

Objects are rarely shaped to allow for mix-and-match composition by a third party – also known as "plug and play." Configuring and integrating an individual object into some given system is not normally possible, so objects cannot be sold independently. Frameworks – the larger units, which are sold – are even worse. Frameworks have traditionally been designed almost to exclude composition. Combining multiple traditional object-oriented frameworks is difficult, to say the least.

Taligent's CommonPoint – the best-known approach that aimed at the construction of many interoperating frameworks – failed to deliver on its promises (although other projects at Taligent did lead to the successful development of new technology). Above all, the approach was overdesigned, aiming for maximum flexibility everywhere, so even the simplest things turned out to be complex. Individual developers were responsible for relatively small parts of the system and thus naturally aimed for *the* solution. The result was an extremely large system for its time. According to a 1995 Ovum report (Ring and Carnelly, 1995), CommonPoint provided over 100 frameworks, covering about 2000 public classes, an equal number of non-public classes and 53000 methods. For comparison, Win32, the Microsoft Windows API, supports roughly 1500 calls. However, the Taligent size has been exceeded by both

Sun's Java and Microsoft's .NET frameworks. The Java 2 Standard Edition, version 1.4, comprises around 2700 classes and interfaces in about 130 packages. The Java 2 Enterprise Edition, v1.3 (which includes the Standard Edition 1.3), comprises around 3900 classes and interfaces (over 5000 when counting CORBA and Apache contributions). The first version of the Microsoft .NET Framework comprises around 4000 classes, interfaces, and types in about 70 assemblies. In comparison to these more recent frameworks, the size of the Taligent effort almost pales, yet it would be fair to observe that these later efforts learned from the Taligent attempt.

Dependencies in such a large system need to be managed carefully. However, the CommonPoint frameworks exposed far too many details and were only weakly layered. In other words, the overall architecture was underdeveloped. For the same reasons, other large industry projects have struggled before, including the major redevelopment effort at Mentor Graphics (Lakos, 1996).

A direct contributor to many of the early fiascos was the chosen implementation language, C++. C++ does not directly support a component concept, so management of dependencies becomes difficult. For example, a fundamental mistake made at Taligent when designing CommonPoint was to assume that the C++ object model would be an appropriate component model, whereas in reality it is too fragile. Blackbox reuse, as introduced in later parts of the book, was neglected in preference for deep and entangled multiple inheritance hierarchies. Finally, overdesign and the C++ template facilities led to massive code bloat.

Merely replacing C++ with another, perhaps cleaner, object-oriented programming language does not solve the problem. A component-oriented approach goes much deeper than simply picking the right language. For example, while some fragilities of C++ are avoided in the design of Java and even more in the design of C#, it is still close to trivial to miss the boat with these languages. Truly component-oriented languages have yet to arrive and even then they will not solve many of the intrinsic engineering tradeoffs that engineers of software components and component software have to face and address.

A project that struggled for a long time is IBM's SanFrancisco framework, which has changed course and reset sails several times over the past years – moving from C++ and CORBA on to Java, and finally to EJB. In its version 2.1 (released in October 2000) the SanFrancisco frameworks counted over 1100 components. In late 2001, IBM stopped SanFrancisco and instead started promoting WebSphere Business Components (WSBC). Then, according to IBM, WSBC was the largest component collection focusing on the server side. WSBC consists of newly designed server-side EJB session and entity components with a focus on financial industries. At least for the part of WSBC that follows the EJB model of strong separation through declared and configurable dependencies, WSBC may now succeed where previous framework approaches failed.

For components to be independently deployable, their granularity and mutual dependencies have to be carefully controlled from the outset. For large

systems, it becomes clear that components are a major step forward from objects (classes). This is not to say that objects are to be avoided. On the contrary, object technology, if harnessed carefully, is probably one of the best ways to realize component technology. In particular, the modeling advantages of object technology are certainly of value when constructing a component. On the flip side, modeling of component-based systems is still a largely unsolved problem, although there have been some recent inroads (for example, D'Souza and Wills, 1999, and Cheeseman and Daniels, 2000). UML, for instance, is still ill equipped to model component-based designs, though improvements in this area are one of the major goals for the UML 2.0 definition (still in its early phases in early 2002).

Under the constant pressure of Moore's law, the world of software has been expanding to take advantage of exponentially more powerful hardware resources. The resulting solutions reach into ever more areas of business and society, leading to new requirements, new markets, and new overall dynamics at a rapid pace. Software technology responded with a very dynamic evolution. However, components are hard to establish in a world of extreme dynamics. While much in the world of software needs to continue to progress rapidly, certain aspects need to evolve more gradually at a more controlled pace. These include the foundations of software components, including component models and basic technologies. The significant market advantage that can be gained from having third-party component backing for any platform creates the necessary feedback loop. To grow and sustain any such third-party communities, the major drivers of software technology, such as Sun and Microsoft, need to maneuver carefully. Players focusing on supplying components can only be viable if investments can be amortized (and profits drawn) before the rules change again. The following chapter covers the interaction between technology and markets in more depth.