# Component Software



The advent of component software may be the most important new development in the software industry since the introduction of high-level programming languages. Component software combines the advantages of custom software and standard software. It enables solutions which are better evolvable, i.e., which scale better, are more readily maintainable, can be extended over time, and can be modernized incrementally.

# 1 Make or Buy?

Today, if you want to solve a business problem by software, one of the first questions you must answer is: make or buy? Do you want to develop the solution from scratch, or buy it off the shelf? The best answer depends on your particular situation and on what is available on the market, but the answer is rarely as clear-cut as you may like.

### 1.1 Custom software

Writing your own custom software has important advantages. You have full control over the software, so you can fully adapt it to the changing needs of your business. This is important, since business needs change all the time: mergers, acquisitions, layoffs, new competitors, new partners, new products, new laws, the Internet; there are many reasons why software needs adaptation to new requirements. If your software better supports your business than your competitor's does, you have a strategic advantage.
However, custom software also has severe disadvantages. It is typically much more expensive than standard "shrink-wrapped" software and its development can take a long time. Long time-to-market is especially critical, since by the time the software is ready, it may already be obsolete, because your business needs have changed again already.

In the last decade, the complexity of software environments has increased tremendously. Highly interactive PCs with extensive multimedia capabilities, networks ranging from local area networks to the Internet, and demanding graphical user interfaces lead to an explosive growth in both operating system complexity and application complexity. To remain competitive, every new release of an application has to support at least some of the new capabilities of the operating system and hardware. This trend leads to ever more features, larger software ("fatware"), longer development cycles, and last but not least, to more defects in the produced software.
To keep up with these increasing challenges becomes harder and harder. Thus it is not surprising that the risks of custom development are avoided whenever possible. Instead, people shop for standard software.
Of course, buying standard software is only possible if at least one vendor has already developed software for this market niche. The more specialized your requirements are, the smaller is your market niche, and the less likely it is that you find an off-the-shelf solution. If there exists no standard solution for your problem, this also affects your competitors. In such a situation, with no standard software to level the playing field, custom software can give you a critical advantage over your competitors.

The financial services market is a good example of where custom software often still makes sense. For example, a bank may develop its own balance sheet rating application, hoping to better judge the risks involved in business loans, and thereby reduce the losses caused by loans that are not paid back.
Figure 1-1 shows the software features that may apply to a given type of market, e.g., for the financial services market, and the percentage of customers in this market who can use these features [Beech]. To the left side, the most commonly used features are located. To the right side, the features desired only by one

customer are located. Custom software works best for products targeting the right side of the spectrum, since there the market is too small for standard software.

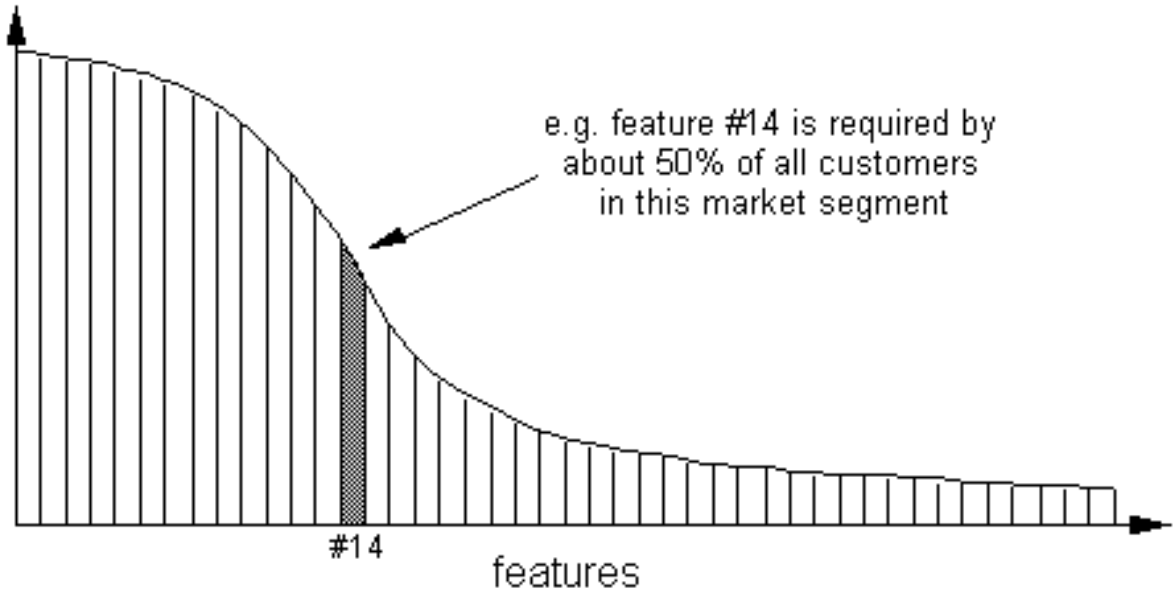


Figure 1-1. Spectrum of required features

**1.2 Standard software**

Standard software is software that you can buy off-the-shelf. Everything you can buy in a store is standard software: word processors, drawing packages, time planners, games, etc. There also exists more specialized standard software for particular types of businesses, e.g., packages for dentists, accounting software, and so on.
Standard software is inexpensive, at least compared to the cost of custom software development. Buying standard software can be done quickly, in particular if you already know which product to buy. Someone may have recommended an application to you, or you may have read reviews and comparisons in journals. This limits the risk you take.

Once you have bought the software, you need to install it. This can be quite a challenge, since many applications have extensive configuration facilities, allowing to fine-tune them to specific needs - up to a certain degree. Some high-end commercial packages go to extremes in this respect. They provide thousands of parameters that have to be set up appropriately. This can be so complex that it is unpractical without the help of expensive systems integrators.
Even though standard software may provide extensive configuration features, it still forces you to adapt your business to the needs of the software, rather than the other way around.

Word processors are a prime example for standard software. Every new generation of word processors adds new features, if only to keep up with the competition. Often, these features are of little use to most buyers, but still everyone must pay for them. Since standard software is sold in large volumes, the costs of the software alone may even be very low. But there are hidden costs, e.g., the cost involved in training, or in upgrading to more powerful hardware. The latter is often necessary because a new software release is typically much larger and slower than its predecessor. Operating systems are further examples of this fatware trend: when your company switched from Windows 3.11 to Windows 95, how much money did it

spend for hardware upgrades or replacements?
Standard software works best for products that address the left side of the spectrum of required features, i.e., where the features required by a large percentage of customers are located. Fatware is standard software that tries to implement too many features that are too far to the right side of its market's spectrum of required features.

# 2 Make and Buy!

Ideally, we would like to combine the advantages of full custom software and standard software: tailored software at the price of off-the-shelf software. Unfortunately, this is asked for too much. Fortunately, there *is* a practical way of getting at least closer to this ideal. The idea is simply to combine custom and standard software. Why shouldn't it be possible to buy 70% of an application in a store, and to develop the remaining 30% in-house? For example, a balance sheet rating application may consist of word processing and rating functionality. Why not buy a word processor, develop the rating algorithm, and then assemble the two pieces into a complete balance sheet rating solution?
Obviously this is only possible if software pieces from different origins can be composed such that they work together in a meaningful way. Such software building blocks are called software components, in contrast to monolithic software.
Most "new" business applications can be implemented as modifications to, or new groupings of, existing components.

### 2.1 What is component software?

It is a fundamental requirement of component software that components may be developed and sold independently of each other, and yet can be combined by the customer. The notion of "selling" is crucial, since obviously you can't buy software components if there is no component market. We will later talk about various technical aspects of components and about industry standards, but those can only be a means to an end. At the end of the day, the only thing that counts for the buyer is the existence of a component market. Definitions of software components should reflect this fact. In particular, we don't consider modular or object-oriented software automatically to be component software. These internal structuring techniques are invisible to, and thus irrelevant for, the only person who can make the whole approach work: the one with money.

The following definition, based on results of the 1996 Workshop on Component-Oriented Programming in Linz [WCOP96], will serve as a basis for the discussion of component software in the next sections.

*A component is a unit of composition with a contractually specified interface and explicit context dependencies only. Components can be deployed independently of each other and are subject to composition by third parties.*

The market-relevant terms here are "independent deployment" and "composition by third parties". The remaining terms are of a more technical nature. We will talk about interfaces in more detail in section 3.2, and about the above definition in section 4.1. Since all dependencies between a component and its context are specified in its interface (see Figure 2-1), the component can be substituted by another one with the same interface. For example, a component may be replaced by a newer version without affecting other software. If interfaces are widely published, independent vendors can start to provide compatible components. An interface thus becomes a standard which creates a market, with competing vendors and with choice for customers.

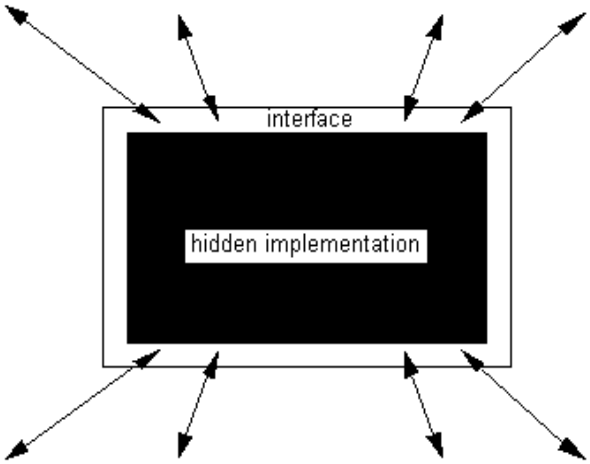


Figure 2-1. A software component is a black box where all interactions occur through a published interface

Given the above definition of components, then *component software is a composition of components, some of which may be standard components and others may be custom components* (see Figure 2-2).

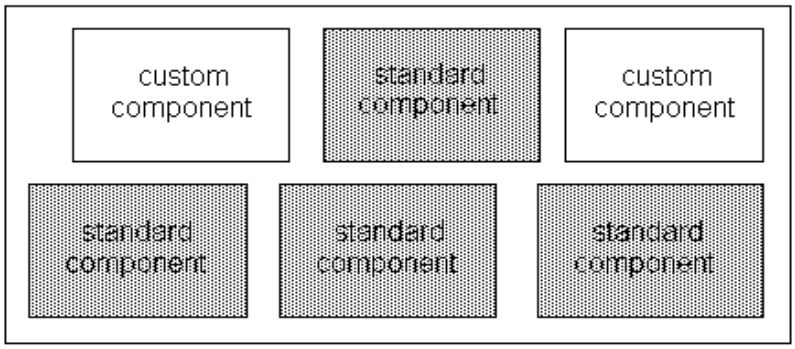


Figure 2-2. Component software as assembly of standard and custom components

This means that component sofware defuses the critical decision between making custom software and buying standard software. Currently we are in a transition period in which old monolithic applications are opened up by adding application programming interfaces - some of them quite general (e.g., using Microsoft COM and OLE), some of them more specialized (e.g., for Netscape Plug-Ins). Microsoft's Office suite is an example of a collection of heavy-weight components, most of them derived from monolithic applications, with all their associated overhead.
More light-weight components are also starting to appear (e.g., ActiveX control objects, Cyberdog objects, or Java "beans"). The full benefits of component software will only be reached if many components become available that are highly focused (and thus relatively small), as this book will demonstrate.

### 2.2 Advantages for vendors

Decomposing software into components follows the old adage of "divide and conquer". By splitting a

complex problem into simpler problems which can be solved independently, the problem becomes more manageable. This software engineering aspect of component software is important for every kind of software, whenever it grows beyond a few hundred lines of code. Of course, the larger and more complex a project is, the more important the modularity of component software becomes. For large problems, modularity also becomes important because it makes team development possible, where different members of a programming team implement different components simultaneously.
Components can be developed in parallel if it is clearly defined in advance what each component should provide, so that a programmer can immediately use the interface of a component currently being developed by another team member - even though its implementation doesn't exist yet. For this to work, it is important that a component's interface be clearly defined and completely decoupled from its implementation, which then can be regarded as a black box. In other words: we don't care what a component looks like internally, in which programming language it is implemented (C++? Component Pascal? Assembler?) or in which programming style it is implemented (procedural? functional? object-oriented? spaghetti-coded?), as long as it is a correct implementation of its interface.
It may happen in a software project that someone remembers that some part of the current problem had already been solved for an earlier project. If this partial solution has the form of a separate component, it can be reused in the new project, thus saving time and cost.
A component implementation can be replaced by a new version if its interface remains the same. This makes it possible, e.g., to send an incremental update to a customer, rather than a new release of the entire application (or operating system). Hence, only a corrected or otherwise improved component needs to be sent to the customer.

So far, the outlined advantages of component software have been mostly traditional software engineering advantages. They are desirable in any type of large-scale software development project. But component software is much more than good software engineering. The main advantage of component software is the creation of markets. Component markets mean that a developer can buy the more generic features of a desired application, and concentrate on the more specific features that make the application truly valuable to the customer. Thus component software lets small developers focus on their core competence and achieve shorter time-to-market. It breaks the trend towards software that is so large and complex that only a few vendors can keep up, while small developers are driven out of business.
If a component is of sufficiently general interest, it can be sold on the market. While this possibly results in a loss of a competitive advantage for the component vendor, it may generate enough license revenues to be attractive.

Looking back at all the advantages we've discussed, we can see that the majority of developers, i.e., small and medium companies, have no business interest in monolithic software. They can only profit from a move towards component software. But interestingly enough, the large companies, which could have more vested interest in sticking to monolithic software, are also shifting towards component software. Probably this is because the maintenance and further development of their monolithic fatware has already turned into a software engineering nightmare so precarious that even unlimited resources won't help anymore.

### 2.3 Advantages for buyers

Component software also has important advantages for its buyers. They get customized solutions quicker. They can save time by buying components on the market, and thereby it becomes less likely that the solutions are obsolete by the time they are ready for use.
Component software allows to add new functionality over time, by adding new components to an already existing solution. In this way, a solution can be extended to handle new needs over time.

In order to use a component, a programmer must have access to its interface description. If the interface is clear and complete, any competent programmer could develop an alternative to an existing implementation.

For example, assume that you have a word processor Write and a spelling checker Spell. The spelling checker allows you to spell-check Write texts, i.e., it uses the Write programming interface. Now, if the better word processor WritePro comes on the market, you want to substitute WritePro for Write, without buying a new spelling checker. This is possible if WritePro implements the same interface as Write does (or a strict superset of it), and if Spell only accesses a text via this interface. In this example, you have replaced one component by another one that implements the same interface.
Replacing an old component by a new one without invalidating other components is an important advantage. It allows to evolve a software system not only by adding new, but also by replacing old components. Outdated "legacy" software becomes less of a problem, because migration to new software can be done incrementally, without losing all investments in existing software.

This is one of the most important characteristics of component software: it replaces "either/or" decisions by more gradual, and thus less critical, decisions. The question is not whether or not to replace existing software anymore; the question rather becomes which components to replace. The question is not "make" or "buy" anymore; the question rather becomes which components to buy and which components to develop. Component software introduces gradual choice where previously there were only hard absolute decisions. Figure 2-3 illustrates the gap in the spectrum between "make" and "buy"; this gap is closed by component software:

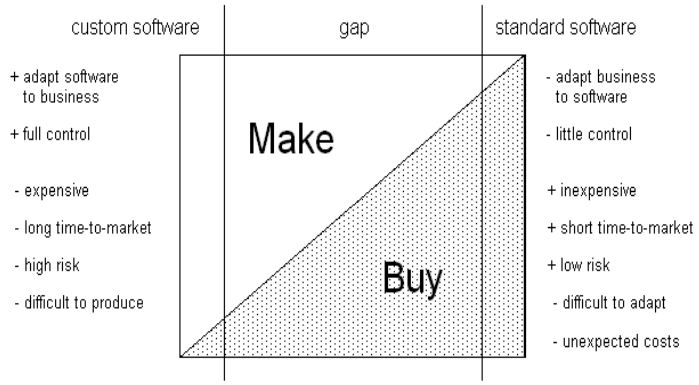


Figure 2-3. Gap between developing everything from scratch and buying everything off-the-shelf

No company will ever be able to develop the best-of-breed products in every possible software category and to integrate them all into one package. Component software allows the buyer to pick and choose the best products among all vendors in the world, and integrate them in a plug-and-play manner. The larger and more diverse component markets become, the more leverage a buyer has in building a uniquely powerful and customized software environment.
For complex problems, buyers may still rely on systems integrators to do the integration of components, but the dependence on single vendors or integrators is much less than it was with either custom or standard software.
Because of this new freedom of choice, the software industry in the long run will likely become much more customer-driven than it is today. Why? Because customers and customer-driven organizations can propose new component interfaces. Such component interfaces, standards in fact, create markets by forcing vendors to compete with each other in creating better implementations of these interfaces.
Components can create a world-wide mass market with fierce competition. The idea of component software is to tap the entire potential of a global software economy, rather than relying on the limited capabilities of only one vendor. Component software simplifies the cross-over of good ideas. After all, even a market leader can steal and reimplement only a few good ideas per new release of a software package; there's just no way he can compete with all the developers on the planet, once they start to build interoperable components. Cross-over makes new good ideas gain market share in an explosive way. Monolithic software with its linear growth of good ideas cannot compete with this advantage of component software.

We have seen that component software is in the interest of small and large vendors, software integrators, and buyers. Everyone wins. But are there domains where component software simply doesn't work?
If we look at the spectrum of required features (see Figure 1-1) it becomes clear that the most obvious candidates for component software are the products in the middle of the spectrum, i.e., features which are desired by many, but not by all, customers. There it makes most sense to create a component market. If everyone had completely unique requirements, there would be no market for reusable components. On the other hand, if everyone had exactly identical requirements, there wouldn't be room for more than a few vendors.
However, even these left and right extremes of the spectrum benefit from component software, due to its inherent software engineering advantages, in particular its better evolvability. But it is more expensive to dethrone established market leaders in these areas.

Today many people still believe that their particular field cannot benefit from component software. It is argued that a domain is too complex, or that there is no market in this domain, and so on. Often, these beliefs come from a lack of experience in the design of modular software. Very complex pieces of software have successfully been turned into components, e.g., whole operating systems. The complexity of a problem does not prevent component software. On the contrary; without a divide-and-conquer strategy such as component software, truly complex problems cannot be handled anymore. Even specialized domains such as embedded systems look like promising opportunities for component software, since there the spectrum of required features looks very good, with many features in the middle of the spectrum. The market in this domain is still underdeveloped, but that doesn't mean that it cannot grow into a strong one.

Yet in spite of all its compelling benefits, component software is no "silver bullet". It doesn't suddenly solve all software problems. Component implementation is still a difficult engineering problem. The design of component interfaces is even more challenging. It takes well educated and experienced engineers to do it. Developing a truly reusable high-quality component cannot be achieved "in one shot"; it requires iterative improvement over a long time, and therefore is expensive. Even pure component assemblers - in spite of customer hopes and vendor promises - won't get very far "without the need for programming".
Component software may abate the software crisis, and it certainly is a necessary and solid foundation for future generations of software. But even component software can only reduce complexity by so much; it is no protection against ever increasing, and often unreasonable demands, put on software.

### 2.4 Compound documents as market catalysts

Once it was hoped that computers will make possible the paperless office. The opposite did happen: producing paper is one of the most effective uses of computers. There are many applications that help producing printable documents: word processors, graphics editors, spreadsheets, graphing programs, etc. A user wants to be able to combine all these types of output into one document. For example, if your favorite mathematical equation editor is not part of your favorite word processor, you still would like to use the equation editor to produce equations and put them into a text written with your word processor. This is a typical case where you want to use the best products of different vendors in a plug-and-play manner, rather than hoping that one single company will be best in all possible domains of document preparation.
From a user's perspective, it is not intuitive that a document being edited belongs to a particular application, and that switching applications may require converting documents between different storage formats. If you are a sculptor, you don't enter the hammer tool, use it, then move the sculpture to the chisel tool, enter the chisel tool, use it, move the sculpture back, etc. That would be a grotesque way to work, but that is how we still use computers today.
Ideally, an editor should merely be a tool to manipulate a particular kind of document contents, such that it can be used side by side with other tools. This results in a document-centric view of computing, where a user opens, manipulates, and closes documents instead of applications. For example, you may open a text

document, insert a picture into the text, click into the picture, and then edit the picture. Such a composition of different types of data in a document is called a compound document. A compound document is a hierarchy of document objects, where some of them may be containers. A container may contain arbitrary other document objects, in addition to its so-called intrinsic contents. For example, a text container may contain pictures, tables, other texts, and so on; in addition to a sequence of characters (its intrinsic contents). The following figure shows a compound document containing text, a table, and a drawing. The table, like the text, is a container. It contains other text and a clock:

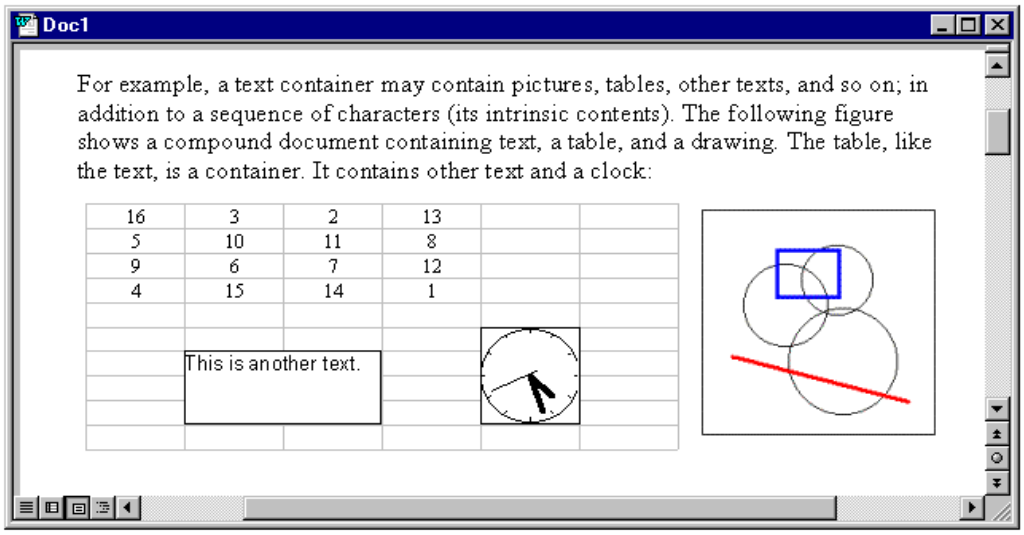


Figure 2-4. Example of a compound document

A compound document is a composition of visual objects. Each type of object requires its own editor, e.g., there must be a text editor for the text objects that are currently displayed. Ideally, it should be possible to add new editors anytime. If this is the case, an editor becomes a software component.
Compound documents imply composition in several ways. Obviously, all objects in a document are geometrically composed in some way. When displayed, they share the window in which they are displayed. When saved, they share the file in which they are stored. The objects also share mouse and keyboard. For this purpose, at any point in time one of the document's objects must be distinguished as current target for mouse and keyboard input. This is called the focus of the document. In Figure 2-5, the focus is visually distinguished by its hatched border:

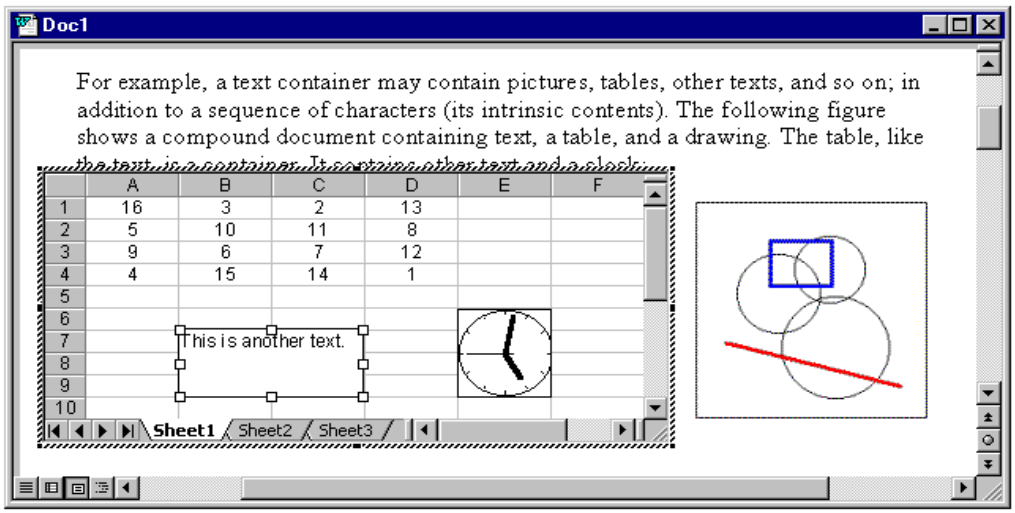

Figure 2-5. The table object of this OLE document is currently focussed

A Web page with pictures, icons, buttons and embedded applets can also be regarded as an example of a compound document. In all our examples so far, texts act as containers for arbitrary embedded visual objects. From here on, we will call such a visual object a "view". Every type of view is implemented by its own software component. Writing texts with embedded views effectively becomes software composition from an end user's perspective.

Compound documents are *much* more general than the term "document" may imply. The views in a document may be active, e.g., they may be running movies or showing a TV channel. They may be interactive, i.e., react on the input of a user. For example, a push button may cause some action when the user clicks on it. Such interactive user interface elements are commonly called controls.
What does a control have to do in a document? A document that mainly contains controls becomes a dialog box or a data entry mask. These things are not usually considered to be documents; but treating them as documents has many advantages. For example, there needs to be no separate dialog box editor ("visual designer", "screen painter"), just a suitable form container. Furthermore, a dialog box form is stored as a document. This is useful, e.g., to later adapt its contents to another language, e.g., from English to German, without needing to recompile anything. In other words: the actual code is largely separated from user interface details such as button locations, text field captions, etc. Figure 2-6 shows the layout of a data entry form, and a second view of the same form in a mode where it can be used, rather than edited:

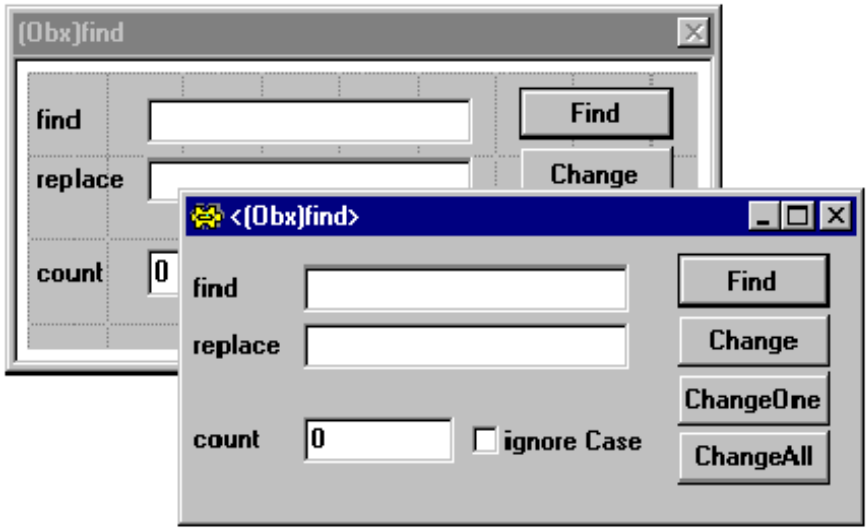


Figure 2-6. Data entry form in a layout-mode view (bottom) and in a mask-mode view (top)

If your application has any kind of user interface at all, it can be implemented with a compound user interface (except for embedded systems with specialized man-machine interfaces such as bar code readers). This means that your component will be based, or at least use, compound documents, even though you might never have thought about your application as being even remotely involved in document processing.

As an example, Figure 2-7 shows the user interface of a logic simulator implemented as a compound user interface. As the ruler and the cursor shape indicate, the control container is a text view. The simulator uses many text editor features, such as tabulators and multiple fonts, or the scrolling facilities of the text container. Implementing such a rich user interface from scratch would be unnecessarily time-consuming. Compound documents allow code reuse by composing user interface elements. Note that the only user interface controls specific to logic simulation are the views which show signal shapes; one of them (Q[0]) is selected. Once the user interface is completed, the container can be switched into another mode where it cannot be

edited anymore, but where it can be used just like any other application.

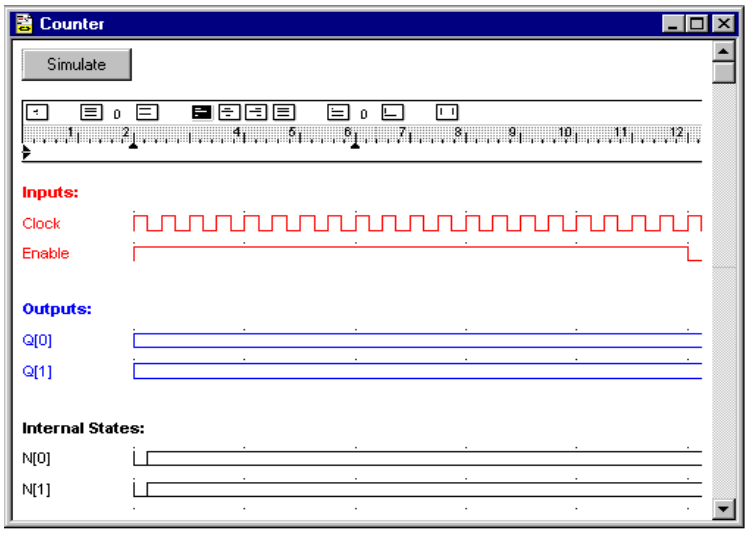


Figure 2-7. Compound user interface of a logic simulator, using a text container and special signal shape controls

We have seen buttons, text entry fields, rulers and signal shape views as examples of controls. A control is a user interface element which cooperates with its container. A control by itself does not make sense; it only becomes useful in conjunction with its container context and the other controls therein. A control is a prime example of a light-weight component completely specialized to a particular functionality. It is only useful when embedded in (i.e., composed with) other components, e.g., text views or form views. A user interface that is built out of controls in a compound document, like the logic simulator example above, is called a compound user interface.

Interactive components which are self-contained, and thus can be used individually as documents or be contained in some other document, are called editors. For example, a 3D drawing or a spreadsheet can be used both stand-alone or as part of another document. Container views are the most advanced kind of editor views.

Composing documents, menus, dialog boxes, and writing the corresponding scripts (or more complex command packages, see below) is what constitutes component assembly. Technically speaking, scripts and command packages can also be components (this is the case with the BlackBox Component Builder), but since such components don't implement objects (i.e., typically contain no classes), we still refer to "component assembly".
Developing components that contain classes which implement visual objects is an example of component construction. Part III of this book will demonstrate component assembly, while Part IV will concentrate on the construction of components. For the sake of brevity, we call components which contain classes that implement visual objects as "visual components". However, be aware that this is somewhat misleading. The component itself has no visual aspects, it is a software package that cannot be instantiated nor displayed - components are not objects.

Compound documents are very intuitive and useful. In fact, they have been so successful in promoting component software that they have almost become synonymous with it. But there also exist many non-visual components such as device drivers. In the future, it can be expected that non-visual components specific to narrower domains will create the largest markets. This may be components for retail, health-care, financial services, embedded systems, and other domains.

One particularly important kind of non-visual component is the "command package" component. The commands of a command package operate on documents, i.e., they interact with visual objects. A spelling checker is an example of a typical command package. Typically, the commands of a command package are represented to the user as menus or dialog boxes. In most cases it is easier to develop command packages than visual components, thus the majority of custom components are command packages.
Simple command packages are often referred to as "scripts". Most scripts are written by end users, rather than professional developers. In table 2-8, a possible taxonomy of components is given.

| **component types** | **description (examples)** |
|---|---|
| visual | |
| control | implements context-dependent interactive objects<br>(command button, check box, radio button, text field, combo box) |
| editor | implements context-independent interactive objects<br>(text editor, spreadsheet, jigsaw puzzle) |
| container | implements context-providing interactive objects, usually special editors<br>(HTML browser, visual designer) |
| viewer | editor with disabled interaction features, i.e., read-only contents<br>(picture viewer, help-text browser, Adobe Acrobat) |
| wrapper | special container that adds or modifies behavior of contained objects<br>(scroller, background-color wrapper, layer wrapper) |
| non-visual | |
| command package | collection of interactive commands that typically operate on visual objects<br>(spelling checker, compiler tool, GUI commands/guards/notifiers) |
| script | simple command package, often for automating work-flow, written by user<br>(receive e-mail -> parse its header -> store it in appropriate subdirectory) |
| library | collection of largely independent functions or classes<br>(math library, string library, collection/tree/bag/... classes) |
| service | provides framework that is extensible through plug-ins<br>(stream input/output, document storage, DB access) |
| plug-in | implements objects that are used by some service<br>(SCSI device driver, Just-In-Time compiler, file converter, DB driver) |
| business | implements domain-specific program logic<br>(accounts/customers/balance sheets, process control configuration interface) |

Table 2-8. Taxonomy of component types

In order to enable any type of component market, a standardized component software infrastructure (so-called "middleware") had to be established first. No one likes to install software that is not immediately useful, thus the middleware had to deliver some immediate benefits to justify the customer's investment. A market catalyst for the necessary middleware was needed.

Because of the immediate and compelling benefits that compound documents provide to end users, they are the means by which Microsoft bootstrapped its component technology into the market. Compound documents have become the main catalyst for the widespread market acceptance of component software technology. Once the technological infrastructure is ubiquitous, non-visual components can also benefit from it.

### 2.5 Component software and the Internet

Decomposing a monolithic program into components should, naturally, result in several smaller components.

Similarly, these components ought to be less expensive than the monolithic program. It doesn't make sense to split up a word processor into a text engine, a spelling checker, a drawing component and a formula editor, if these components together are more expensive than the monolithic program. For this reason, it can be assumed that mass market standard components will be inexpensive. Of course, the smaller a market, the higher prices may be.
Selling a component involves overhead, e.g., for packaging and marketing. For an inexpensive component, the overhead may be larger than the selling price, i.e., no profit can be made. For this reason, components today are usually sold in packages; e.g., a dozen Visual Basic controls may be packaged and sold together. This is unfortunate; it would be much more appropriate to sell components individually.

This problem may be solved by the Internet. The Internet is a low-cost distribution medium for all kinds of digital things, why not for software components? In fact, already there are the first examples of "software kiosks" on the Internet, where you can buy components and receive them electronically. If successful, electronic distribution will enable even the smallest vendors to sell components in a cost-effective way, since no intermediate distributors are involved. Moreover, distribution is always world-wide, making the Internet an unprecedented market accelerator. At the time of this writing, there have already been several software kiosks on the Web, e.g.:

http://java.wiwi.uni-frankfurt.de
http://www.buydirect.com
http://www.broadcast.com
http://www.componentsource.co.uk
http://www.cybout.com/cyberian.html
http://www.devdepot.com
http://www.partbank.com
http://www.pparadise.com
http://www.software.net
http://www.stream.com
http://www.unboxed.com

Table 2-9. List of software kiosks

Even with modems getting faster all the time, no one wants to transfer megabytes of data over the Internet if it can be avoided. Thus fatware is ill-suited for the Internet; components should be as small as possible. Huge applications have been accepted in the past, since disk and memory sizes have increased so much. But development tools that create minimal-sized "components" of over 2 MB, where 2 KB would be ample, are hardly tolerable anymore. The Internet is a good reason to strive for component software in the first place, and for small components moreover.

The first Web pages have been rather static, but Java applets are beginning to change this. Applets are components that are sent over the Internet to a Web browser and execute there. Therefore, making the Web experience more active and lively is another driving force towards component software. Component software benefits from the Internet as distribution medium, and the Internet benefits from component software to make it richer and more interactive.

Compound documents have been mentioned as the most important market catalyst for component software technology. This is the case because of the sheer number of personal computers in use today, which give an unprecedented economy of scale. But enterprise networks in large companies have been a second catalyst that helped finance the necessary investments into middleware that might be useable for component software. In large enterprises, the main application of such technology lies in adding programming interfaces to old legacy applications, such that they can be connected to enterprise-wide networks in a simpler and more systematic way than previously possible. Unfortunately, this has led to the misconception that

component software and communication between objects that reside on different machines ("distributed objects") are one and the same [OHE96].
But many distributed object systems need to be deployed as a unit in order to work, because they are tightly coupled - even though the coupling occurs over a network. Their parts are not independently deployable components, and thus those systems are not component software systems. On the other hand, Microsoft's OLE is an example of true component software that doesn't involve distribution over a network.

Distributed objects are just another, and possibly becoming an important, application of component technology. But most components, in particular light-weight components, only interact locally.
Today, distributed objects are often hailed as *the* future of computing. In our opinion, the vision of distributed objects is greatly oversold. Distributed objects hide the most important characteristics of a distributed system, in particular a tremendous difference in speed (latency caused by network round-trips) and in error probability (every method call may fail due to a network problem). As a result, complete local/remote transparency, probably the most attractive aspect of distributed objects, cannot be sustained in practice. Replacing complicated network interfacing code, typically based on the widely used "sockets" API, is the major benefit of using distributed objects today. Less humble benefits, such as complete location transparency, or unlimited scalability through decentralization of resources, remains to be proven in practice.
It is clear, however, that large enterprises can benefit from any technology that makes integration of applications a simpler and more systematic process, and that they have the money to create an attractive market. In fact, the most interesting future component markets will probably be for business components on the server side, rather than on the client (PC) side where the margins are much lower and the requirements more generic. Whether networked business components will be achieved with distributed objects or with other middleware technologies remains to be seen. For example, messaging middleware may become more popular again, because the messaging model is simply more adequate, easier to use (e.g., simpler error handling), and more flexible (e.g., multicasts instead of mere point-to-point communication).

The Internet, with its World-Wide Web, combines aspects of both compound document and network architectures. Thus it is a combined market catalyst that consequently has become the third major driving force for the adoption of component software.

Today, some possible future effects of the Internet can only be guessed. The Internet not only allows to distribute low-cost software, it also allows free distribution of software. Free software can make sense if the development expenses can be recouped by selling special services for this software, or by introducing some pay-per-use scheme. The former scheme basically treats software as a marketing vehicle for services. The latter scheme requires a trustworthy accounting mechanism. Obviously, only features far to the left side of the spectrum of required features can be expected from free software; the other features are usually of too little interest to the companies or individuals that develop free software.

# 3 Requirements for Component Software

The idea of component software is so obvious that we may wonder why it didn't catch on earlier. To answer this question, and to give some first technical insight on what component software is, we will first take a look at the (pre)history of component software. We will see that an important reason for the late appearance of general component software was the lack of an appropriate and standardized infrastructure.

### 3.1 Operating systems

In the early days of computers, an application program had to take control of the entire computer. The application could freely use all hardware resources, there was no other software around on the same machine at the same time. However, this also meant that the program had to implement from scratch even

basic services such as input and output routines.
Later, generally useful services were packed together into a special program that was always available, the so-called operating system. An application program now didn't have complete control over the entire machine anymore, it had to cooperate with the operating system and use its services where available. However, this loss of control was more than compensated by the increased productivity that resulted from reuse of the operating system services.
An operating system can be regarded as a software component that is reused by all applications written for it. Seen the other way around, an application can be regarded as a software component that dynamically extends its operating system's functionality. The separation of software into operating systems and application programs can be seen as an extremely successful first example of component software.

Decades ago, computers were very expensive devices. Thus it was hardly acceptable that one single program should block the entire computer when it runs. In particular not for interactive applications, where most of the time, the program would simply wait for the connected user to type something on a terminal. Thus time-sharing was invented, i.e., the operating system allowed to execute several applications simultaneously, by giving each application a short time-slice before passing control to the next one. Since computers had become fast enough, users wouldn't notice that their computer only worked part-time for every one of them.
This was a big step forward in giving more people access to computers. But it also created problems. A program should not be able to interfere with other running programs. In particular, it should never be allowed to change the state of another program, i.e., to overwrite memory belonging to another program. Of course, a well-behaved program wouldn't do that. But how about incorrect programs? Even then, it was clear that few programs are free of errors and thereby immune against vicious behavior. Thus safety became an important concern.
The problem was solved by adding hardware protection mechanisms. Hardware protection could be combined in an elegant way with support for more flexible memory management, so-called virtual memory. Basically, every program is presented with the illusion to have the computer's entire memory for itself, and this memory is separate from the memory that other programs use. These separate address spaces prevent any direct interference of application programs. A program could directly interact only with the operating system, via special kernel calls. Modern microprocessors have special support for such kernel calls.

Unfortunately, hardware protection works all too well. It prevents interference of applications, but it also prevents integration. This is one reason why component software could not evolve on this basis. Tight integration is necessary for component software to work! A rating algorithm must have direct access to the balance sheet it is rating. Over time, operating systems added features to support some degree of application integration, such as pipes or shared libraries. However, these mechanisms have been far too limited and inefficient. They didn't provide an appropriate basis for componentizing operating systems and applications, and thus failed to stop the fatware trend.
The one area in which the old operating systems *have* been successful concerning component reuse is device drivers. Device drivers have a venerable tradition of isolating closely cooperating components, i.e., the device drivers, the applications and the operating system, from each other's details. This is done in a manner that is efficient whilst still allowing a measure of independent evolution of all components involved.

In order to write an application, a developer needs to know what services the operating system provides. He doesn't necessarily need or want to know how these services are implemented. For this reason, abstract descriptions of operating system interfaces have been created. In order to enable portability of applications, and thus to open the greatest possible market for them, the interface of an operating system had to be the same independent of implementation or hardware variations. The MS DOS operating system has shown how successful strict adherence to a standard can be, while Unix has shown how fatal slight variations of the same theme can become. Component users want shrink-wrapped software that works in a plug-and-play manner, rather than bother with a variety of incompatible binary formats.

To write an application, it isn't sufficient to know what functions an operating system provides. In order to create actual code, the developer - or at least the compiler writer - needs to know down to the last bit and instruction how a call must be made and how the parameters must be passed; i.e., there must be a binary standard for the calling conventions. Over time, different calling conventions have been tried out. Some operating system even require the use of several different calling conventions, e.g., the Mac OS uses half a dozen different conventions. However, most calling conventions have proved too limited in scope. For example, if a microprocessor only supports a few thousand different kernel calls, in practice these calls can only be used for calling the operating system from an application - and even the operating system may outgrow the processor limitations. Only the more modern calling conventions can be generalized to calls between an arbitrary number of independently developed components, including (but not limited to) the operating system.

An operating system must be able to load and start applications dynamically at run-time. This requires a standard file format for the code of an application, so that the operating system's loader can perform its work. Like the calling conventions, these file formats (e.g., the Windows EXE file format) have turned out to be too limiting, and new formats more amenable to component software were defined (e.g., the Windows DLL format).
To summarize the most important points concerning component software that can be learned from operating systems, the following list of requirements for component software can be derived:

• a standard that enables dynamic loading of components (dynamic link libraries, calling conventions)
• a standard programming interface (e.g., the Unix kernel interface)
• a protection mechanism which prevents a component from illegally modifying the state of other components
• a way to share data between components without copying them back and forth, and without explicit conversions to or from linear byte streams

We will meet these points again and discuss them in more detail in the following sections.

### 3.2 Interfaces as contracts

A component is a black box that interacts with its environment only via its interface. The interface defines a standard for what component vendors have to provide and what component users can expect. If the interface is published, many vendors and users can take advantage of it, and thus a market can develop around it.
We can look at the interface of a component at the binary level, the domain-specific application level, and sometimes at the user interface level. But before looking at existing standards for these various interface levels, we should take a closer look at what an interface is.

An interface defines what a component vendor must provide, and what a customer can expect to get. For example, a mathematics component may implement an interface that defines procedures for calculating the sine and cosine functions:

```
DEFINITION Math;

    PROCEDURE Sin (x: REAL): REAL;     (* return the sine of x *)

    PROCEDURE Cos (x: REAL): REAL;     (* return the cosine of x *)

END Math.
```

As Bertrand Meyer [Meyer89] has observed, an interface can be compared to a contract. In this section, we will demonstrate that many aspects of genuine contracts can directly be applied to component interfaces.

Contracts involve at least two parties, e.g., a vendor who promises to provide some goods, and a customer who promises to consume the goods in a particular way. For example, an author writes a book, and a book publisher publishes this book. Both author and publisher are bound by a contract.
Sometimes a contract comes from a third party. For example, many professional organizations offer standardized contracts for their profession.

There are good contracts and bad contracts. A good contract should be clear, complete, and concise. A bad contract is ambiguous, misses important points, or lays down irrelevant details. All these deficiencies lead to one party making false assumptions about the behavior of the other:
If a point in a contract is ambiguous, each party must supply its own interpretation of what its and the other one's duties are. This easily leads to incompatible assumptions, and then to conflict.
If something important is missing, tacit assumptions may pop up. If such an assumption remains valid for some time, it may turn into an unwritten law, i.e., into an implicit refinement of the contract. But this refinement is fragile, since the assumptions may suddenly not hold anymore, when the other party sees reasons to do things in a different way in the future. This leads to a power struggle or cancellation of the contract.
If irrelevant details are fixed in a contract, an unnecessary constraint is put on one or both parties. There comes the time when one party wants to change the contract in order to be more flexible. This can lead to expensive and possibly futile attempts at renegotiation. Or instead of negotiating, a contractor may simply violate the contract. If this leads to no complaints, the violation also may turn into a kind of established right. But power struggles, cancellation of the contract, or litigation are likely.
In summary, contracts that are ambiguous or underspecified are fragile, while overspecified contracts are too constraining. Both are likely to lead to conflicts. Interestingly, the conflicts often break out when the contract needs to be amended for some reason. This is a very important effect, as we will see later. The art lies in defining contracts which are neither under- nor overspecified.

Contracts have other interesting properties. For example, if someone has to provide a certain amount of valuable goods, the receiver usually won't complain if he receives a larger amount. Vice versa, if the receiver for some time requires a smaller amount than agreed upon, the provider usually won't mind either. This means that in some cases, contract violations are harmless. Another interesting point is that contracts often have an expiration date, i.e., they are only valid for some predetermined period of time.

What does this all have to do with interfaces between components? The parties bound by a contract correspond to components interacting through some interface. Bad contracts are unclear, incomplete, or overloaded.
An unclear contract corresponds to an unclear interface specification. Unfortunately, this is rather the rule than the exception. While the syntax of an interface can be defined easily, clear semantics are elusive. Usually there is just an informal text describing what the interface means. Formal and semi-formal specification methods can help to make interfaces less ambiguous. For example, a procedure may specify preconditions and postconditions, i.e., which conditions must hold on the inputs of the procedure when it is called, and which conditions on the outputs must be established by the called procedure. This goes a long way towards making interfaces less ambiguous, although it still doesn't address all important aspects, such as performance.
For example, the following formal specification

```
PROCEDURE Sum (x, y: INTEGER; OUT z: INTEGER)
    Precondition: (x >= 0) & (y >= 0)
    Postcondition: z = x + y
```

indicates in the precondition that the input parameters x and y must not be negative. In the postcondition, it is specified that the result parameter z is the sum of x and y.
This is an interface (contract) between the implementor of the procedure and the caller of the procedure. The

implementor is free to accept weaker preconditions, e.g., it may also accept negative values for x and y. While this can never be wrong, no one may rely on the assumption that negative values for x or y are handled in a particular way. In a similar vein, an implementor may support stronger postconditions, i.e., it may deliver more than what is required.
An example of a legal implementation of the above specification is the following:

```
PROCEDURE MySum (x, y: INTEGER; OUT z: INTEGER);
BEGIN
        z := ABS(x) + ABS(y)
END MySum;
```

which is a perfectly correct implementation of the interface specified above. It is very typical that an implementation is much more specific than its interface requires: the above example implements a weaker precondition (negative values are also permitted) and a stronger postcondition (sum of absolute values is computed) than required by the specification.

An interface which defines too many inessential details will cause programmers to use them and to rely on their availability. This makes it impossible to change these details later, even though it may become strongly desirable to do so. Giving too many details is especially enticing if there already exists some complex but undocumented code, which is to be turned into a component. Then the easy thing to do is to publish the source code and not bother with the definition of a less constrained interface. As a result, the entire implementation becomes the interface and may never again be modified, since this may break the client code, i.e., the other party.
Using a complete implementation as its own specification is also a problem of complexity: it is hard to analyze tens of thousands of lines of code to determine how a piece of software behaves. Such a complex "specification" is similar to a contract that contains a book full of small print.

Even if a software component has a minimal and well-defined interface and is not available in source code, an inventive programmer will find out by trial and error how the component behaves under circumstances that are not mentioned in the interface. Basically, such a programmer derives his own ad-hoc interface specification that is more specific than the published interface, i.e., the contract. If he takes advantage of this more useful but fragile extended interface, then his code may break when the component is replaced by a new version whose internal workings have changed. We all know this effect from applications that suddenly don't work correctly anymore after a new operating system release is installed. These applications relied on assumptions that were not written in the contract and thus not guaranteed. Of course, if a sufficient number of important applications rely on such undocumented features, the operating system vendor may be forced not to change them in the future anyway. Thereby the vendor would, grudgingly, accept the established rights of the marketplace.
The problem of ad-hoc specifications derived by trial and error, and the fragility caused by them, cannot be solved completely. But, as we will see later, there are popular programming language constructs that make it much more likely to fall into this trap. Consequently, they should be avoided in component interfaces.

Can interface contracts be enforced? To some degree, such an enforcement is possible. Some contract violations can be detected at compile time by the compiler, other violations can be detected later at run-time, by using suitable hardware and software protection mechanisms.
At the most basic level, every component is required to interact with other components through their interfaces exclusively; this is a kind of universal contract, i.e., a law. For example, directly overwriting another component's memory would be a gross violation of the law. In a closed world, e.g., in an isolated monolithic application, this problem can be solved easily. But in an open component software world, interference between components is a fundamentally more critical issue. If the hardware or software infrastructure of a computer can completely prevent the violation of interface contracts and laws, the reliability of the whole system will improve, by limiting the damage that a component can possibly create ("bug containment").

There exist hardware protection facilities and in particular modern programming language designs that provide this kind of safety.

When a component is sold worldwide, the vendor typically doesn't know all its customers anymore. This means that it has a contract with unknown parties. Since contracts can be changed only if all parties agree, it follows that an interface of such a component may never again be modified. Except in the harmless ways described earlier, i.e., by providing more, or by requiring less than what was specified in the original interface.

In principle, it would only be possible to withdraw an interface, i.e., to cancel a contract, if a timeout were specified in the interface. This is unusual today. But since the market is changing so rapidly and thus software becomes obsolete so quickly, interface timeouts don't seem so essential today. But who knows; this may change in the future.

Finally, if a vendor and a customer agree on using a standard contract from a professional organization in their field, this is similar to a component constructor and a component assember who agree on the same standardized component interface that was defined by some third-party framework designer. Framework design will be discussed in more detail in 4.2 and in Part II.

We have seen that contracts give a deep insight into the nature of component interfaces. By now, we should have become sensitive to the problem of good interface specifications. Unclear interfaces lead to incompatibilities, underspecified interfaces lead to the invention of ad-hoc interfaces by trial and error, and overspecified interfaces provoke overly constrained or outright incorrect implementations. Underspecification and overspecification are both latent sources of conflict: when a component is replaced by a new version, dependent components may suddenly break.

### 3.3 Object models

An object model defines the necessary rules to make components compatible on a binary level, such that components can interact on a particular machine even if they have been developed independently. To achieve this goal, an object model needs to standardize the following aspects:

1) an interface definition language for describing interfaces
2) a mechanism for inquiring about interfaces and their attributes, i.e., an "interface repository"
3) a code file format for storing code that can be loaded at run-time, i.e., a format for dynamic link libraries
4) a mechanism for locating suitable code files for a given interface, i.e., an "implementation repository"
5) a mechanism for loading code files into memory, i.e., a linking loader
6) a mechanism for version checking of loaded code
7) a mechanism for creating instances of a loaded class, i.e., for the allocation of objects
8) calling conventions for methods of an object
9) a mechanism for navigating between polymorphic types of an object
10) a mechanism for reclaiming unused memory, i.e., for the deallocation of objects
11) message formats, if distributed objects are supported

These points will be discussed one by one in the following paragraphs.

1) A component may use the services of another component. To achieve this, a developer only needs to know the used component's interface. Possibly, there does not even exist an actual implementation for this interface yet. An interface needs to be described in some way. A textual description of an interface is written

in an interface description language (IDL). Ideally, this is just a subset of the programming language that the developer uses. However, a general object model is language-independent, and of course an IDL can only be a genuine subset of very similar programming languages. Other programming languages will exhibit a more or less severe degree of "impedance mismatch", i.e., a translation is required between interface constructs. For example, an IDL may define unsigned integers, while languages like Java or Component Pascal only support signed integers. This requires a mapping, e.g., to auxiliary data types defined in a library. This approach may become extremely invonvenient.

2) An IDL description provides information about an object's interface to the developer. If possible, a compiler should use the same information to check whether the interface is used correctly. For this purpose, there often exists a binary format for interface descriptions in addition to the textual IDL format. The available collection of such binary descriptions is called the interface repository. It may just consist of a collection of so-called symbol files, or it may be stored in some kind of database.

3) When a component is compiled, the compiler needs to create a file containing the generated code. The format of such a file must be suitable for run-time loading, i.e., it must be a dynamic link library. Often, the operating system's DLL format is used. For language-specific object models, a more efficient light-weight DLL format can be used instead.

4, 5) At run-time, when a component for the first time needs the services of another component, this component is loaded. The loader first locates the code file of the component. Locating the code file may be straight-forward, e.g., the name of a component may directly determine the file system path to its code file. Alternatively, a configuration database may be consulted to determine the correct location of a suitable code file. This indirect approach is more flexible, but is more demanding in terms of system administration. The collection of code files or the corresponding database is called an implementation repository.
It has been proposed to develop even more flexible ways to search a suitable component implementation for a given interface, using additional search criteria. Whether these so-called trader services will succeed in practice remains to be seen.

6) When the loader has successfully located a component and loaded its code into memory, it can check whether the loaded code really implements the interface that was requested, and whether the versions of the loaded component and its client are compatible. This check is of fundamental importance, because it is not acceptable that a version conflict leads to a crash some time later, leaving the user without clue as to the source of the problem. Some object models provide no adequate versioning mechanism and shift the burden of consistency checking partially or completely to the client component.

7) Once a component's code has been loaded and checked, instances of its classes can be created. For this purpose, the object model needs to define an allocation mechanism for objects. Some object models suggest an indirect approach to allocation, in order to gain additional flexibility. In particular, an object may be created by another object, a so-called factory object. A factory object may decide on its own how to allocate or how to initialize objects.

8) When an object thus finally has been created and made accessible to others, its methods can be called. A method call is a procedure call performed indirectly via an object, so that different objects can lead to different code being called. Typically, an object contains a pointer to a table of procedure pointers. Each element of the table corresponds to one method of the object. A method call then simply becomes an indirect procedure call via the object's method table. Usually the calling conventions of the underlying operating system are used for these procedure calls.
Fortunately, this has the effect that an object model can be used by every programming language that supports operating system calling conventions, plus references to objects or references to functions ("procedure variables") - i.e., by practically every modern language, whether object-oriented or not.

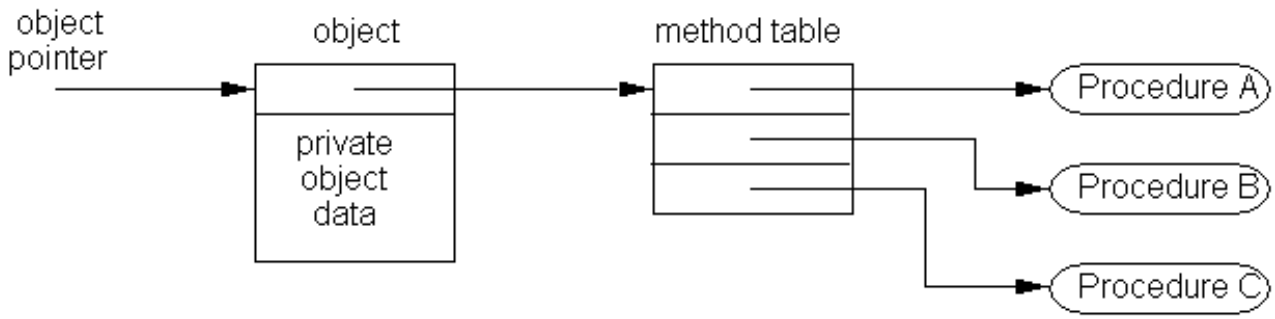


Figure 3-1. Possible memory layout of an object

9) Polymorphism is one of the fundamental properties of any object-oriented or component-oriented system. In a program, some interface supported by an object may be known at compile-time. This is called its static type. But an object may reveal additional capabilities, i.e., an extended interface, at run-time. Depending on the object's dynamic type, these capabilities may differ. This is called polymorphism ("many shapes"). An object model needs to provide a means to gain access to these optional capabilities if and only if they are available. An object-oriented programming language defines language constructs such as type extension (also known as subtyping or interface inheritance), type tests, or type guards (i.e., safe type casts) for this purpose. The object model must provide similar functionality. Without polymorphism, a system would not be extensible.

10) When an object is no longer used, i.e., when it is no longer being referenced from the outside, it must be deallocated to free the memory that it occupies. In a component world, the objects that a component makes accessible to the outside may become referenced by any number of other objects that it doesn't even know about. There must be rules that establish who must deallocate a given object, and when. For example, the object that releases the last remaining reference to another object could deallocate it. To determine whether an object owns the last reference to another one, a mechanism is needed that helps tracking references, e.g., a reference count in each object which counts the number of currently existing references to it. When the count goes down to zero, the object's memory can be freed. Correct usage by all components is critical for such rules to work. In a closed application, incorrect memory management is one of the most expensive sources of errors; but at least you know who to blame if the application crashes. In an open component world, one malfunctioning component can cause others to crash, which makes it difficult to pinpoint the culpable vendor. Thus memory management in a component software environment is a fundamentally more critical issue than for monolithic software. Ideally, the rules and mechanisms defined by an object model should be sufficiently simple, complete and clear that they can be automated; i.e., it should be possible to relieve the developer from manual deallocation, by providing an automatic garbage collector. Automatic garbage collection in an open world is no luxury, it is a necessity.

11) An object model that supports distributed objects must define a message format, which describes the byte streams produced by a remote method call. The caller of a method is called the client, the callee is called the server. In the most general scenario, the server object is implemented on another machine than the client. From a developer's perspective, the client can directly call a server object's methods. In reality, the client only interacts with a proxy object, which is a local representation of the true object implementation on the remote server machine. In most implementations, a proxy, as well as its server-side counterpart, can be generated automatically out of the object's interface.
A remote method call causes the proxy to send a request message with the input parameters to the server (see Figure 3-2). On the server, the genuine method call is executed. Its results are sent back in a response message. Both request and response messages are linear byte streams; they represent the values of the method's input (request) and output (response) parameters. Using a distributed object model to replace traditional means of communication can be attractive under certain circumstances. Here it is interesting to

note that transparent distribution of objects fundamentally requires the complete separation of interface from implementation: a client object must not be aware whether it is calling a genuine object, or only its proxy. The interface is the same in both cases, but the implementation is radically different.

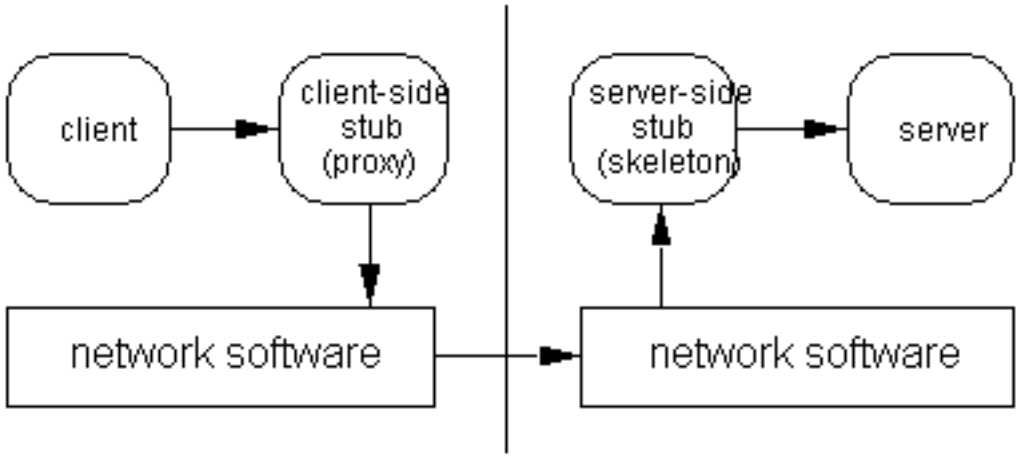


Figure 3-2. Remote method call

Now we have seen all major ingredients of an object model. An object model makes it possible to replace one component by another one, e.g., by a new version, without forcing a recompilation of any other component. This is called release-to-release binary compatibility. For example, if you buy a new release of your favorite word processor, you don't want to be forced to buy new versions of your spelling checker and of all other components that operate on texts. The problem solved by release-to-release binary compatibility is also known as syntactic fragile base class problem.
Using a new version of a component must not lead to a system crash just because, e.g., its method table entries have been rearranged. All aspects of an object model must fit together in a way that makes release-to-release binary compatibility possible. Solving this problem is a prerequisite for the asynchronous evolution of components, which is inevitable in a large market.
After this overview over what an object model is, we will take a look at some of the more important industry standards for object models and how they address the issues discussed above.

### 3.4 Standards for object models

IBM's System Object Model (SOM) has been introduced in OS/2 mainly as a means to make the operating system extensible by new components; not so much as a means to split applications into components. Over time, SOM has been extended to conform to the CORBA (Common Object Request Broker Architecture) standard. CORBA is a standard of the Object Management Group (OMG, http://www.omg.org), a consortium consisting of hundreds of companies. The OMG attempts to define various standards for distributed objects. CORBA standardizes the general architecture of an object model, while leaving open all machine-specific ("binary") aspects. Compatibility is only required in two areas: for source code of applications, and for the message formats on a network. On the binary level (e.g., calling conventions), CORBA products are free to use completely different approaches. Theoretically, this approach will still allow object models of different vendors to work together.
CORBA provides a simple API that is available for all objects. Additionally, server objects have access to an extension of this API, to the services of a so-called *object adapter*. In principle, there may exist different object adapters; each one optimized for particular kinds of server objects. The mandatory basic object adapter, which is the only one widely available, is optimized to manage relatively few, but heavy-weight objects. This can be explained with the background of most CORBA customers; namely large enterprises

that are interested in wrapping object interfaces around existing monolithic legacy applications. For most server object implementations, it is necessary to access product-specific extensions of the basic object adapter's interface. Moreover, it has become obvious that to achieve more convenience, language-specific object adapters would be useful. Unfortunately, object adapter implementations are not portable, because they have to be implemented in terms of a product's private, non-standard APIs.
CORBA defines an interface definition language (IDL) in which the syntactic interface of an object is specified, independent of the programming language used for its implementation. Functionally, the SOM extension of the CORBA IDL is a mixture of C++ and Smalltalk features. It supports multiple inheritance and metaclasses, but leaves out templates and garbage collection. Multiple inheritance allows to combine several existing interfaces into a new one. In contrast to CORBA, which does not support implementation inheritance, SOM supports (multiple) implementation inheritance. In contrast to CORBA, SOM also supports normal pointers instead of reference objects only. Typical C++ sources that use or implement SOM objects rely on SOM-specific macros that are not common in other CORBA implementations, and thus limit their portability.
SOM allows to extend a class in a new release, e.g., by adding new methods, without rendering existing clients of the old class incompatible, i.e., it solves the syntactic fragile base class problem.
CORBA defines interfaces for interface and implementation repositories. CORBA is self-describing, i.e., its own interfaces can be obtained via the interface repository. These features are supported by SOM.
SOM uses the calling conventions and DLL formats of the operating system on which it is implemented. A SOM method call is a procedure call with two additional parameters, one for the self-parameter (the object) and one for the environment pointer (used e.g., for exception handling).
A SOM object provides version numbers (major and minor), such that a client can check whether it accesses a legal version of an object (unfortunately, checking is usually done at class load time, which is not sufficient - an object may later be passed unchecked to some other object requiring a different version). It is possible to test at run-time whether an object is an instance of a particular class or one of its subclasses.
Distributed SOM (DSOM) is a CORBA-compliant extension of SOM that adds support for distributed objects.
Like CORBA in general, SOM makes no provisions for the systematic reclamation of free memory, i.e., it is not possible to develop a garbage collector that automatically frees all unused memory. Apple's Mac SOM implementation differs in this respect and provides reference counting methods in the SOMObject base class. Unfortunately, this is a non-standard programming model that cannot be relied upon when writing cross-platform software. For example, OpenDoc, which is based on SOM, provides its own redundant reference counting scheme for some of its objects. On the positive side, Mac SOM is considerably more efficient than IBM SOM and does not exhibit the memory leak problems of the latter.
SOM is available on several platforms besides OS/2, e.g., for AIX, OS/400 and Mac OS. SOM, Orbix (Iona), NEO (Sun) and other CORBA implementations implement different subsets of the CORBA services, a collection of CORBA libraries ranging from support for persistent objects and generic data structures to transaction and security services. However, these standardized libraries are only partially implemented and most vendors offer competing proprietary libraries which are more agressively marketed and better supported.
DSOM was set to become the first widely available CORBA implementation, because of the support by IBM (OS/2) and Apple (Mac OS), and because it is the basis of OpenDoc. It didn't happen. Now the hope of CORBA supporters lies with VisiBroker of Visigenic. This product is licensed by Netscape, Oracle, Novell, Borland, and Sybase. It is the second, and probably the last, opportunity for a mass-market CORBA implementation, and thus for component software based on CORBA.

Microsoft's Component Object Model (COM) was designed as a means to integrate applications, and will be the basis for most new operating system services of Windows. Microsoft's interface definition language (MIDL) for COM is an extension of the older DCE (Distributed Computing Environment) IDL. DCE is a standard for remote procedure calls.
A component may provide a so-called type library, which is its interface repository. The Windows registry is used as a simple implementation repository. The Windows calling conventions are used for COM. A COM code file may either be a Windows DLL, for light-weight components that are loaded into the same address

space as its client; or as Windows EXE files, for heavy-weight components that have their own address spaces (typically normal Windows applications).
The COM model differs from CORBA in that an object may provide not only one, but an arbitrary number of interfaces - to each of which there can exist arbitrary numbers of external references. A client only has access to an object's interfaces. Thus it leaves completely open how an object is implemented, whether as one programming language object, or as a composed data structure. In the simplest case, a COM object provides one interface and is implemented as one contiguous memory block. This situation is shown in Figure 3-3. Note that COM is a binary standard, meaning that the memory layout of an interface, the calling conventions, and the code file formats are completely determined by COM (actually, the code file format is not determined by COM itself, but by Windows and the chosen processor's instruction set format).

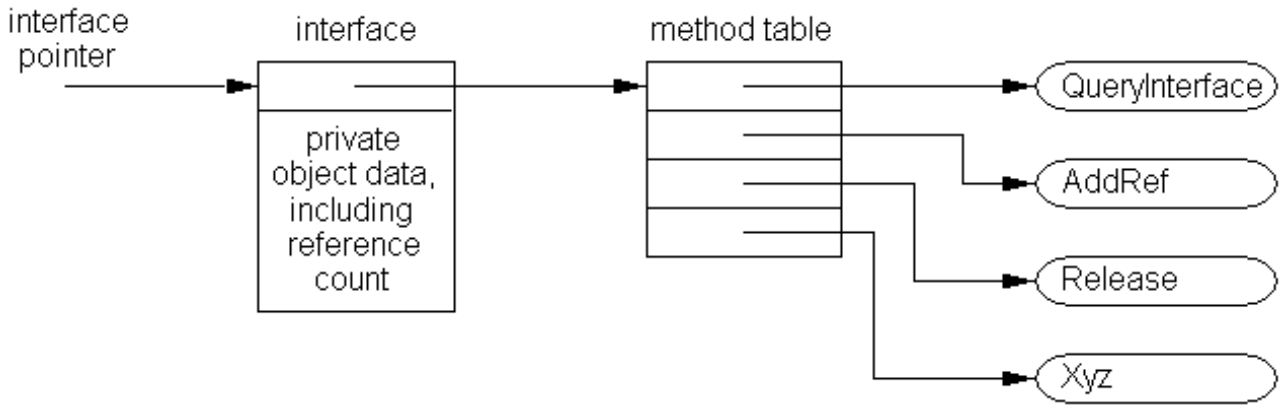


Figure 3-3. Layout of a simple COM object

More typical is the situation illustrated in Figure 3-4: the object consists of a data structure containing many internal (light-weight, non-COM) objects and a few interfaces. These interfaces can be referenced from other COM objects.

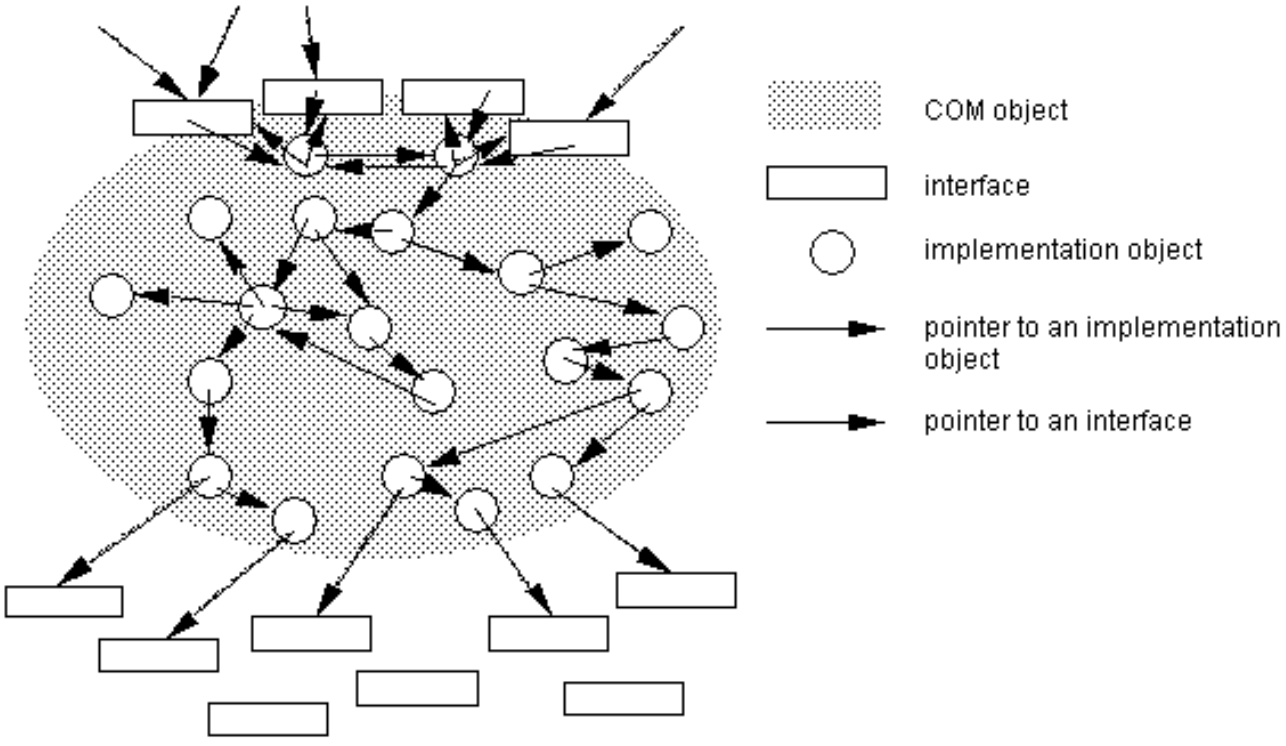

Figure 3-4. A COM object may provide several immutable interfaces, and typically consists of many internal objects

Every COM interface provides the method QueryInterface. It allows to navigate between COM interfaces and is the way COM achieves polymorphism. In the diagram above, there exist pointer paths between all provided interfaces; they are necessary to implement QueryInterface. QueryInterface also acts as a version checking mechanism: if it returns a particular interface, its corresponding functionality is available. A syntactically or semantically different version would have to be provided via another interface. To distinguish interfaces, every interface gets a globally unique identifier (GUID), i.e., a 128-bit number that is unique in space and time.
COM objects are allocated indirectly via factory objects. A COM object that is implemented as a DLL, or as an EXE that registers itself upon startup, is called an ActiveX object.
COM addresses the syntactic fragile base class problem by making interfaces immutable, i.e., once officially introduced, an interface may never be changed again. This also implies that the interface's binary layout (the ordering of methods in its method table) will never change again. If it is necessary to extend an interface, the new functionality must be provided as a new, additional interface. If the new interface is a strict superset of the old one, it can be regarded as a subtype. For this purpose, COM supports (single) interface inheritance in its IDL and type libraries.
The earlier discussion of interfaces as contracts helps to explain why a COM interface is immutable: once it is widely published, an interface cannot be changed anymore without unilaterally breaking the contract that it represents.
Unlike the usual version numbering schemes, the COM scheme of immutable and uniquely identifiable interfaces makes it possible for several independent parties to create new versions of an interface, without risking to create version number conflicts. Furthermore, it simplifies a migration from old software to new software; by allowing to support old and new interfaces at the same time during the transition period.
Concerning memory management, COM uses reference counting: every COM interface contains, in addition to the QueryInterface method, the methods AddRef and Release. AddRef must increment a reference count, while Release decrements the reference count. When the reference count goes from one to zero, the last external reference vanishes, and the interface can deallocate itself, and possibly other data of its object if it is the only existing interface to it.
Distributed COM (DCOM) is an extension of COM that adds support for distributed objects, using a simplified implementation of DCE as its basis. DCOM treats client and server symmetrically in that a client doesn't know whether it calls a client-side stub (proxy) or the real server, and a server doesn't know whether it is called by a server-side stub or a real client.
COM has also been ported to other platforms than Windows, e.g., to Mac OS and several flavours of Unix [COM]. COM started as a proprietary standard, but later Microsoft passed control to the Open Group standardization organization.

Like CORBA, but in contrast to SOM, COM does not support implementation inheritance. This decision has a subtle but important reason. Like an ambiguous or underspecified contract, an inheritance interface forces programmers to rely on assumptions, which may not hold anymore for a new release of the base class. This is called the semantic fragile base class problem. Implementation inheritance can only be fully controlled if the base class' source code is available, i.e., if the implementation is used as its own interface. But this is similar to an overspecified contract with pages of small print, in that it prevents later improvements of the base class implementation.
Microsoft correctly argues that implementation inheritance doesn't pose problems within a component, which is a white box from its implementor's point of view, but that it should not be used across components. Using implementation inheritance across components would require strict rules that help avoid the semantic base class problem. Unfortunately, these rules are still a research topic [Szyperski97]. Consequently, class libraries for component software, e.g., the SOM OpenDoc classes, avoid implementation inheritance even if their object models support it. Interface inheritance, also called subtyping, is not affected by these problems

since it has nothing to do with implementation aspects. This topic, and the terms involved, will be discussed in more detail in section 4.1.
If you hear someone from IBM complaining that SOM has solved the fragile base class problem long before there was COM, this person refers to the *syntactic* fragile base class problem, i.e., to release-to-release binary compatibility. On the other hand, if you hear someone from Microsoft talking about the fragile base class problem, this person refers to the *semantic* fragile base class problem. This kind of misunderstanding is so stereotypical it is almost funny.

SOM and COM are examples of language-independent object models. An object model is always part of the run-time environment; but it may be designed to support one language particularly well. This leads to language-specific object models, e.g., for Java. Language-specific object models have several advantages; in particular, their IDLs can be pure subsets of the language, which is the most natural and convenient way to develop software.

Sun Microsystems has defined an object model for its Java Virtual Machine (JVM). Java was designed to support components downloaded from the Internet. Since it is impossible to reliably judge the quality of most software on the Internet, safety against malfunctioning or even hostile components (viruses!) is a prime concern. For this reason, Java is a typesafe language, and adherence to its safety rules is not only guaranteed by the compiler, but also by the JVM class loader. For example, it is impossible to perform unchecked type casts in Java code. This rules out the use of unsafe languages such as C, C++ and even original Pascal. These languages cannot be compiled into JVM code (otherwise there would have been no fundamental reason to switch from C++ to Java in the first place). In contrast, JVM compilers for Smalltalk, Eiffel, Lisp, Component Pascal, or a garbage-collected version of Ada are feasible.
Since dangling pointers would make Java unsafe, the JVM requires a garbage collector. The precise mechanism to be used is left open to the implementation.
Java code is compiled into Java byte code, which can be regarded as the instruction set of a virtual Java processor. The JVM either interprets the byte codes or it precompiles them into native code at load- or run-time, using a so-called Just-In-Time compiler (JIT). Class files are completely machine-independent, so that unlike typical SOM or COM components, pure JVM code is machine-independent. Each class is compiled into its own class file. Usually the file system is used as a simple interface and implementation repository, in the form of a collection of class files.
Java defines a concrete code file format, but leaves open the main memory representation of a program. This is just the opposite of COM: COM is not so concerned with the code file format, but predefines a concrete memory representation of an object's interface and its method table. CORBA leaves abstract both disk and memory representation.
Version checking of Java code is done at run-time. The existence of a method is checked upon its first call. This is a questionable approach, because a component may be loaded and work correctly for a while, but suddenly - of course when you can least afford it - an unexpected exception occurs.
(A similar problem can arise when you obtain a COM interface via QueryInterface, but one of its methods returns an error code meaning that it isn't implemented. For example, we have not seen any OLE container yet that implements the IOLEContainer.EnumObjects method for the iteration over a container's embedded views.)
Java distinguishes interfaces from classes. Interfaces can be related by multiple interface inheritance (subtyping). A class is an implementation of one or several interfaces; classes can be related by singe implementation inheritance (subclassing). Checked type casts allow to navigate between different interfaces supported by an object. Semantically, this is equivalent to the mechanisms supported by CORBA and COM.
The distinction between interfaces and classes would lend itself to minimizing the semantic fragile base class problem: if subclassing is used only within components, and interfaces are used for interface inheritance within and across components, then the semantic fragile base class problem is largely avoided. Unfortunately, Java library designers don't seem to follow any clear guidelines on when to use interfaces and when to use classes.
For a distributed implementation of Java objects, there exists Java RMI (Remote Method Invocation).

Alternatively, distribution can be achieved via CORBA or COM, requiring language mappings between these object models and Java. Language mappings define, e.g., how unsigned CORBA or COM integers are mapped to signed Java integers (Java does not support unsigned integers in the language).

Compatibility and safety on a binary level are the most important areas where JVM differs from SOM and COM. Safety has many aspects, the most fundamental being memory integrity. About half of all programming errors are in this domain: illegally overwriting memory, not freeing unused memory (memory leaks), or prematurely freeing memory that is still used (dangling pointers). CORBA and SOM do not address this basic issue at all, which is a monumental failure. Without a memory management standard at the object model level, component interoperability and reliability are severely endangered. COM defines a reference counting mechanism, which solves the problem at least for non-cyclic data structures. Unfortunately, most interesting data structures are cyclic, and thus COM pushes some of the complexity of memory management to higher levels of software. Reference counting alone cannot reliably prevent memory leaks, but fortunately it can prevent the more dangerous dangling pointers. JVM provides true automatic garbage collection and other safety features.

Component Pascal is a language in the Pascal, Modula-2 and Oberon language family. It is a component-oriented replacement for these languages. Its safety properties are similar to those of Java. For interfacing to third-party software and for writing device drivers, there are special libraries that allow to develop unsafe code in a controlled manner. In many respects, Component Pascal's object model is similar to Java's. You will meet its most important aspects in Part III of this book. A brief history of Component Pascal, and a summary of the differences between Pascal and Component Pascal, are given in the appendix.

A special Direct-To-COM compiler version supports COM in that it directly maps Component Pascal records to type-checked COM interfaces (including checks of the MIDL parameter modes). Most importantly, this special compiler relieves the programmer of manually handling COM's reference counting mechanism, i.e., it adds a garbage collector for COM objects. Since this compiler only automates what would have to be done manually anyway, it doesn't incur additional overhead.

| **Aspect** | **CORBA** | **SOM** | **COM** | **Java** | **Component Pascal** |
|---|---|---|---|---|---|
| originator | OMG | IBM | Microsoft | Sun | Oberon microsystems |
| IDL | CORBA IDL | ext. CORBA IDL | ext. DCE IDL | Java subset (1) | C.P. subset (1) |
| meta classes | no | yes | no | no | no |
| pointers (2) | no (3) | yes | yes | yes | yes |
| implem. inheritance | no | yes | no | yes | yes |
| interface repository | yes | yes | type libraries | yes (4) | yes (5) |
| binary memory standard | no | yes | yes | no | no |
| binary code file standard | no | no (native DLLs) | no (native DLLs) | yes (portable DLLs) | no (native DLLs) |
| portable code files | no | no | no | yes | no |
| implement. repository | yes | yes | registry | yes (4) | yes (6) |
| version checking | no (7) | no (7) | immutable interf. | yes (8) | yes (9) |
| calling conventions | undefined | native | native | special (10) | native + special |
| run-time type system | yes (11) | yes | yes (12) | yes (13) | yes (14) |
| memory mgmt. | undefined | undefined (15) | ref. counting | garbage collection | garbage collection |
| syntactic fbcp solved | no | yes | yes | yes | yes |
| semantic fbcp addressed | no impl. inh. | not addressed | no impl. inh. | interfaces/classes (16) | by convention (16) |
| distributed objects | yes | DSOM | DCOM | Java RMI (17) | no (18) |

(1) CORBA and COM access possible through language mappings or "direct" compilers, plus interfacing libraries
(2) can pointers/references of a programming language be used directly, e.g., for method calls or parameter passing?
(3) auxiliary reference objects must be used, which reduces convenience and efficiency
(4) direct mapping of class files is typical
(5) direct mapping of symbol files is typical
(6) direct mapping of code files is typical
(7) there exists minimal support for the client to do the checking (pragmas, major/minor version numbers)

(8) automatic check when feature is used for the first time
(9) automatic check per object when the module is loaded
(10) native calls require wrapper classes as glue; efficient in-line calls not possible; cannot create DLLs with native calls
(11) interface repository, get_interface(), InterfaceDef, is_a, describe_interface
(12) QueryInterface, IDispatch, IProvideClassInfo
(13) type tests (instanceof), type casts, Java Reflection Interface
(14) type tests (IS), type guards, Meta module
(15) except Mac SOM, which uses reference counting
(16) by convention, implem. inheritance is only used within components (Java libraries are inconsistent in this respect)
(17) object marked as remote object by subtyping from a special library interface
(18) DCOM supported as backplane object bus, via a special Direct-To-COM compiler

Table 3-5. Comparison of various object models

Which object model will win? Probably none. It is likely that COM and some CORBA implementations will coexist for a long time to come. Bridges will connect the COM and CORBA worlds. This will cause many interoperability problems, but they might not be much worse than working with several different CORBA implementations, or with several programming languages and one CORBA implementation, simultaneously.

Microsoft has recently announced to extend COM with support for automatic garbage collection and exception handling. While remaining basically language-independent, Java-like languages should profit from such special support. Sun on the other hand has introduced the Java Native Interface (JNI), which uses a method layout similar to the one of COM. It looks like the fight for object model predominance concentrates more and more on COM and JVM, which begin to converge to some degree. CORBA could be reduced to a secondary role as an invisible communication layer that can, but need not, be used by the other two object models.

We have only talked about the basic object models, which are the low-level plumbing necessary for interoperability. However, object war may finally be decided by the services built on top of the object models. Probably not by the services for fine-grained objects, such as the CORBA relationship or persistence services, but by the services that are required for reliable cooperation of medium-sized components. In a desktop environment, these are compound document, database access, and networking services. In an enterprise environment, these are security services, asynchronous multicast messaging services, transaction services, and directory services. CORBA has standardized these services, but actual software for some of them is only appearing now. The several years of lead that CORBA enjoyed has turned into a head-to-head race with Microsoft, and Java not far behind.

A major question concerning CORBA, which is a committee design influenced by hundreds of companies, is whether it will suffer from the same problem as Unix, i.e., whether it will also remain an eternal "almost standard". Whom will enterprise customers trust more: a consortium with a mixed record of cooperation, or Microsoft with its sometimes monopolistic attitude?

Today (September 1997) it seems most likely that CORBA (e.g., IBM's ComponentBroker, which replaces the ill-fated DSOM) will remain an important infrastructure for the communication between enterprise applications for some time. Its Internet Inter-ORB protocol (IIOP) may still become a preferred communication protocol for distributed Java objects (via Java RMI). All the rest of CORBA, including the IDL, language bindings, and in particular the object services, have largely failed in the (component) market place. It is unrealistic to still hope that CORBA will play a role as an effective (component) object model with a large market of binary-compatible shrink-wrapped components.
Microsoft has won on the desktop, makes inroads into the enterprise market thanks to Windows NT, DCOM, Internet servers, and its transaction service. Microsoft also embraced Java as a language and integrated its run-time system closely with COM.

It is possible that over time, developers will realize that general object models like SOM or COM are

important as "backplane object buses", but that their direct use is often too expensive. Frameworks may provide bridges to simpler and more efficient object models. These light-weight and cost-effective object models will typically be language-specific and may not directly support distribution. There is a precedent for this development: there exist industry-standard hardware backplane buses, e.g., Motorola's VME bus, that are expensive, complex and relatively slow. But today there exist high-volume, low-cost VME boards that are relatively generic (frameworks) and provide a local, simple, and fast low-cost extension bus (Industry Pak bus) for which there are several hundred extension boards available on the market. A similar development is likely to happen in the software world also. Note that once a backplane object model is available, it is much less expensive and risky to establish one or several more focused secondary object models on the market, since they don't need to be supported universally. For more exotic requirements, you can always fall back to the backplane object model.

Alternatively, someone may come up with an object model which is language-independent, but still light-weight enough to allow efficient development without cumbersome separate IDLs and inefficient cross-process communication. Microsoft is attempting this feat with a major new generation of COM, called COM+. COM+ is an in-process object model; it still uses DCOM for cross-process and cross-machine communications. Compared to COM, it has a more abstract definition of how objects and classes are represented in memory (and on disk). It can be regarded as a more general architecture than the Java virtual machine, since it supports a broad spectrum of programming languages and code file formats. In contrast to COM, it provides a systematic way to extend its own run-time infrastructure by third-party components (e.g., debuggers, profilers, etc.) in a standard way. It eliminates the reference counting problems of COM by providing true garbage collection. Its types fully support Java interfaces and classes (unfortunately including implementation inheritance and field access across components...). Extensive use of meta data makes COM+ a very dynamic environment, well suited for Java, Component Pascal, Visual Basic, scripting languages (whether compiled or interpreted), and it obviates the need for implementing the cumbersome and redundant IDispatch interfaces of OLE Automation. Many standard services, including security and transaction services, are planned for COM+. They are similar in scope to the CORBA object services, but more focused and better integrated.
If Microsoft makes no major mistakes, the possibility to mix COM and COM+ components should make transition relatively painless and virtually guarantees the success of this new standard. The technical advantages are compelling: a standard object model which
• is as flexible as possible, but as concrete as necessary to allow binary compatibility of shrink-wrapped components (although: whether giving up a standard memory representation for object interfaces is worth the additional complexity, and what it means in terms of efficiency, remains to be seen)
• is a virtual machine architecture for dynamic languages like Java (no need for IDLs or language-mapping code anymore), including scripting languages
• provides a standardized infrastructure for third-party low-level tool components
• supports garbage collection to eliminate the unsafe and cumbersome manual life-cycle management of objects
• supports efficient in-process cooperation of different components in a safe way

It can be hoped that COM+ helps to overcome the unfortunate Unix process concept which has so long stood in the way of useable component software, to eventually make place for single-address space operating systems which achieve protection through safe programming languages and possibly even through innovative hardware protection schemes. Meanwhile, it will be a challenge to use the new COM+ (i.e., Java) features in a way which does not result in maintenance and administration nightmares, caused by semantic fragile base class and version management problems.

### 3.5 Standards for compound documents

Which kinds of services should a software infrastructure for compound documents provide? Basically, it's all

about resource sharing. Views in a compound document share screen space, files, input devices, etc. There must be rules for how views must cooperate when using shared resources. Without such rules, a view may draw outside of its own area, may overwrite the contents of another view on a file, and so on.

Rules can be put down in a contract, i.e., they can be cast into interfaces. For such interfaces, the most successful standard so far is Microsoft's OLE (originally an abbreviation of Object Linking and Embedding). OLE defines an architecture for the visual representation of compound documents. It is complemented by architectures for storing OLE objects (Structured Storage) and for scripting OLE objects (OLE Automation).
These architectures are embodied in a large number of COM interfaces, e.g., for persistent storage of views, for drag and drop between views, for in-place editing of views, and much more. Mostly for historical reasons, OLE makes a strong distinction between different kinds of views, so-called document objects and control objects, and similarly, between document containers and control containers. Also for historical reasons, there is a distinction between the normal object model (i.e., COM) and a special object model for Automation, i.e., for component assembly. Visual Basic is a typical so-called scripting language that uses Automation.
Unfortunately, these distinctions burden the developer with unnecessary decisions and add considerable complexity. For example, Microsoft recommends dual interfacing, i.e., to provide every interface both as a normal COM interface and as an Automation interface. A COM interface is efficient and powerful but possibly unsafe, while an Automation interface is safe but less efficient (and does not support exactly the same parameter types). Java has shown that a language-specific object model can combine sufficient power and safety in one object model; and Component Pascal has shown that this need not even lead to a performance penalty.
OLE was originally designed to allow different traditional applications, e.g., Microsoft Word and Microsoft Excel, to cooperate. For this reason, most OLE-enabled software are complete applications (EXE files). Light-weight components implemented as DLLs only appeared later, in the form of ActiveX control objects. Control objects share the same address space, while an application that implements a document object resides in a different address space than the object itself.
One reason for the success of OLE is that it eases the transition to component software by supporting both large traditional applications and light-weight components, although the otherwise redundant support for EXE files causes another considerable increase in complexity.

OpenDoc was designed by the Component Integration Labs (CILabs) as an alternative to OLE. The CILabs were a consortium consisting of Apple, IBM, and hundreds of other companies. OpenDoc was designed from scratch to support true components. As a result, OpenDoc views, so-called Live Objects, need not be designed both as components and complete applications at the same time. Another difference to OLE is that OpenDoc was designed as a cross-platform architecture from the outset. Like OLE, OpenDoc provides an automation interface in addition to its SOM object model (OSA, for Open Scripting Architecture), which is equally redundant and cumbersome to implement as the OLE automation interface. As a positive aspect, there is no technical distinction between document and control objects.
OpenDoc is a framework consisting of SOM classes. It barely uses implementation inheritance, and no multiple inheritance at all, and thus should not suffer much from the semantic fragile base class problem. OpenDoc provides some services that in OLE are provided at a lower level, in fact already in the COM object model. In particular, an extension mechanism which allows to dynamically obtain additional interfaces of a view (similar to QueryInterface), and a reference counting scheme for memory management.
An OpenDoc document constitutes its own address space, which is shared by all OpenDoc views that are part of the document. A view's implementation may reside in another address space, using DSOM. Like OLE, OpenDoc's handling of address spaces is a compromise between putting everything into one address space (very flexible, convenient, efficient; but highly unsafe) and putting every component in its own address space (safe, but very inefficient and cumbersome). A more satisfying compromise could only be achieved through better hardware protection facilities (unlikely to happen) or by relying on safer programming languages. More on this later.

OpenDoc seems simpler and less burdened by backward-compatibility concerns than OLE, although it has to

make up for some deficiencies of its underlying object model. In particular, OpenDoc provides a much clearer notion of what a view is, and controls and containers are simply special views. OLE, however, has no clear notion of what constitutes a container; different containers may support different subsets of container interfaces. This may result in a considerable fragility, since OLE objects will have to be tested in as many containers as possible, with no guarantee to work in others. It would be very helpful if Microsoft published a clearer and stronger contract for containers in the future.
The varying capabilities of ActiveX containers highlight a general danger of the COM (and Java) programming style: if too many interfaces of a service are optional, it becomes exceedingly difficult to guarantee correct behavior of a client under all circumstances, because nothing can be taken for granted. This effect is dreaded by programmers who have used Microsoft's ODBC (Open Database Connectivity) library, where almost every procedure may fail, because its implementation is optional.
Incidentally, the same headache occurs with distributed objects, where a server or network breakdown may lead to failure of any method call. Transactions are the only practical escape from this problem, but they are beyond the scope of this book, and shouldn't be mandatory for locally interacting components anyway.

To help achieve a high level of component consistency and quality, the CILabs provided validation suites and validation services for OpenDoc parts. This was a laudable initiative that should have been a great boon to developers of compound document components, in particular containers, and buyers looking for quality components.
However, OpenDoc never gained critical mass in the market. Delays of the Windows version, shifting commitments of the CILabs member companies, and the Apple management's lack of understanding of the nature of component software finally made OpenDoc fail. It was always treated as a fringe technology, instead of something fundamental to the future of computing.

In spite of OpenDoc's demise, some of its ideas may still turn up in future versions of Java Beans. Java Beans is a framework that allows the implementation of portable controls, or the wrapping of native controls into Java classes. Java Beans is not a full-fledged compound document architecture, it is missing the generality necessary to develop advanced containers. Moreover, bean containers are restricted to be immutable, i.e., they may not resize, add or delete embedded beans. This reflects the immutable nature of an HTML Web page.
Today, Java Beans is only a layer between Java components and full compound document architectures such as OLE or OpenDoc. It is most likely that in the future, Java Beans will be extended to become a full-fledged, stand-alone compound document architecture in its own right.
Java Beans shows an interesting design quirk. It asks bean clients not to use Java's type tests on beans; instead, a bean provides methods which are similar to COM's QueryInterface. The fact that both Java Beans and OpenDoc had to introduce a QueryInterface-like mechanism in addition to Java's and CORBA's type systems indicates that these type systems may not be the ultimate wisdom.

The BlackBox Component Framework, which will be discussed in more detail later in this book, is similar to OpenDoc in many respects, but more light-weight. In contrast to all other architectures, it provides extensive support for containers, the most complex kind of view. It hides the difference between the OLE and OpenDoc user interfaces, so containers can be written that automatically look and feel correctly, whether running under OLE or OpenDoc. A BCF's container mode is a powerful feature that makes every container a visual designer, as we will see in the tutorial parts of the book.

| **Aspect** | **OLE** | **OpenDoc** | **Java Beans** | **BCF** |
|---|---|---|---|---|
| Name of a component | ActiveX Object | Live Object | Bean | BlackBox Component |
| Creator of standard | Microsoft | Apple | Sun Microsystems | Oberon microsystems |
| Owner of standard | Open Group | CILabs † | Sun Microsystems | Oberon microsystems |
| Object model | COM | SOM | Java | Component Pascal |
| Automation | Automation | OSA | Java | Component Pascal |
| Compound files | Structured Storage | Bento | yes 1) | yes 2) |
| Multiple file formats per view 3) | yes | yes | no | no |

| | | | | |
|---|---|---|---|---|
| User controls choice of editor | yes 4) | yes 4) | no 5) | no 5) |
| Multilevel undo/redo support | no | yes | no | yes 6) |
| Menu sharing | yes | yes | planned | yes |
| Data interchange/conversion | yes | yes | planned | yes |
| Imaging | Windows GDI 7) | e.g., QuickDraw 7) | Java AWT | BCF |
| Properties | yes | no | yes | yes |
| Overlapping views | yes | yes | no | yes |
| Non-rectangular views | yes | yes | no | no |
| Embeddable in editor | yes | yes | no 8) | yes |
| Container iterators | in principle 9) | no | no 10) | yes |
| Container user interface | OLE | OpenDoc | not available 10) | OLE / OpenDoc 11) |
| Container modes | no | no | limited 12) | yes 13) |

1) There is language and library support (metaprogramming/reflection) for internalization/externalization of beans.
2) Module *Stores* supports internalization/externalization of arbitrary graphs, including pointer translation.
3) Meaning that an object may support different file formats and possibly convert between them.
4) There are end user tools which allow to define a prefered editor for each file type.
5) The type of an object is stored along with its data (automated with metaprogramming facilities).
6) Several undoable commands can be composed into a script which is undoable as a whole (nested undo).
7) OLE and OpenDoc assume use of the underlying operating system's drawing libraries.
8) While being used ("run-time"), a bean container cannot be edited (i.e., moved, resized).
9) Iterator support for containers is optional, and implemented by practically no one.
10) A bean has no container support.
11) Module *Containers* supports general containers, while abstracting from OLE and OpenDoc user interface details.
12) Only "design-time" and "run-time" can be distinguished. At design-time, beans can be moved and resized.
13) Editor/Layout/Browser/Mask modes make special visual design or documentation authoring tools superfluous.

Table 3-6. Comparison of various compound document architectures

In the last section, it was mentioned that it makes sense to complement a universal backplane object model with more specialized light-weight object models. This approach can already be observed with Java. Java Beans is a framework that can be used as a bridge between the JVM object model and backplane object models and compound documents, in particular to COM/OLE. BCF follows a similar strategy for Component Pascal components, but with a considerably more comprehensive framework than the current Java Beans.

### 3.7 Standards for domain interfaces

Standards for object models and compound documents are important, but by no means sufficient to fully exploit the potential of the component software market. For this to happen, more vertical, i.e., more specialized and domain-specific standards are necessary as well. Components targeted at different domains cannot interoperate even if they use the same object model. For example, it doesn't make sense to embed a COM-based network driver in an OLE document, even though OLE is also based on COM. Their domains are simply too different. For integration to make sense, there must be some common ground, some common context. This context is described in an interface. Remember that the definition of a component says that all its context dependencies must be specified explicitly in its interface.

In can be expected that over time, more and more specialized standards will be developed for domains such as financial services, process control, Internet communication, database connectivity, and so on. Today, we are only at the beginning of this process, leading to more and more specialized component interfaces.
But first examples do exist. There is a group of companies that has defined OLE for Process Control (OPC); Apple has defined interfaces for Internet OpenDoc views, its Cyberdog view collection is a first implementation of these interfaces; Java Database Connectivity (JDBC) is a standard interface for Java components that need to access relational databases; Commerce API is a Java API for secure financial transactions over the Web; OLE for Retail Point-of-Sales (OPOS) drives peripheral devices such as bar code

readers; Retail Application Framework Technology (RAFT) supports three-tiered retail applications; and so on.

We think that some of these initiatives will be successful, others will fail (all OpenDoc initiatives have failed already). Creating standard frameworks for "business objects" is a very challenging technical, political, and financial undertaking. But once a domain standard has reached a critical mass of support, the economy of scale and the choice that it affords cannot be beaten by a proprietary approach anymore.

Probably, domain-specific standards will increasingly be defined by user organizations rather than by vendors, since they require more domain knowledge than vendors typically have. Today's market is driven by vendors, but tomorrow's market will likely be driven more by customers.
The most important thing about a component interface is that it opens up a potential new market. It creates a battle ground where component vendors can and must compete with each other. New competitors may be attracted; established companies may loose their dominance.
The timely creation of a good interface standard is critical. A lengthy standardization process can result in a standard that is obsolete by the time it is completed. Hasty standardization can result in a low quality design that needs to be revised or replaced, causing loss of time and money. But there is no way around some kind of standardization, because without it, there will be no component markets.

# 4 Development Tools

Component software is a good idea, but without components, it doesn't work. Components must first be created, and this requires suitable development tools. And even if components exist, there must be tools for their assembly. Some tools may be specialized to component assembly, others to component construction, and some will be suitable for both assembly and construction. Even fewer products will be good also for the development and evolution of entire new component frameworks.
The market for assembly tools will be at least a factor ten larger than the market for construction tools. In this chapter, we will look at the issues raised by component software concerning programming languages, framework, and development environments. Finally, we'll take a first look at the BlackBox Component Builder, the tool that will be used later in the book.

### 4.1 Languages

Which programming languages can be used for developing components? The short answer is: almost any language will do. However, we also attempt to give a longer and less superficial answer to the question in the following text.

Basically, most languages today either look like a variant of C or Pascal, e.g., C++ and Java look similar to C, while Ada, Component Pascal and even Visual Basic and Eiffel look similar to Pascal. Lisp, Smalltalk and some other "exotic" languages constitute a comparatively small minority. For example, a 1995 poll of over 400 university courses has shown that about 65% still used a Pascal-style language, about 16% used a C-style language, and the rest was distributed among other languages.
As long as you use a C- or Pascal-style language, you can assume that a sufficient number of developers with the necessary skills is available. The cost of learning another language of the same style, e.g., Component Pascal when you know Pascal, is negligible compared to the cost of learning today's huge libraries and operating system interfaces.
This means that there is considerable freedom to choose a suitable language for a given job.

In practice, language-independent object models have much increased this freedom lately. Most languages today are able to access language-independent object models such as COM. The object model adds the

dynamic features that the language may lack. Language-independent object models make language decisions less strategic than they used to be: deciding on one language for one component doesn't preclude using another one for another component. Using a particular language doesn't create an automation island anymore. Consequently, there is no reason not to use the best language for the job.

Does a language make a difference at all, considering that coding is only a small part of a project's overall cost? In fact, minor syntactic issues, e.g., how a loop construct looks like, don't cause any measurable difference in a project's cost (assuming that the infamous GOTO has more or less died by now).
Thus a language comparison around "programming in the small" doesn't make sense anymore. But a modern programming language is much more than just a notation for the implementation of small algorithms. A well-designed programming language also supports programming in the large. To remain manageable, large programs must be decomposed into components that interact via specified interfaces only. A good programming language can be used not only as an implementation language, but also as an interface definition and specification language. Interfaces define the *architecture* of a system: those parts that a developer can rely on; the static properties of a system that must not be violated; the structural backbone that supports the used components. A language that allows to express more of a system's architecture explicitly ("statically") in its notation for interfaces, makes it possible to write tools that help ensuring the consistency of an implementation with its specification. A compiler can flag violations of an interface at compile-time, when it is still inexpensive to correct the problem. Run-time checks make it possible to detect other interface violations as early as possible during testing.
The static expressiveness of a language, and the tool support enabled by it, becomes even more important when interfaces are modified, which is often necessary in a design and prototyping phase, or later in the maintenance phase of the software (which is responsible for about 80% of the total development cost!). In fact, the architecture of a large software system invariably deteriorates over time, when modifications or extensions are made. Improving the architecture of a system, by eliminating old baggage and streamlining a subset of its interfaces and components, is called refactoring. Today, this is one of the most neglected design issues. But component-oriented languages are powerful refactoring tools that reduce the time, cost and risk involved in changing parts of an existing system.
This means that a state-of-the-art programming language *can* make a difference in most phases of a software component's life cycle, and in the life cycle of a component software system (which can be much longer than the life cycle of any of its components)! Thus, the common cliché that the choice of programming language is irrelevant is based on too narrow an argumentation that misses the dimension of "programming in the large".

Programming languages that support explicitly formulated interfacing constructs, e.g., types, variable declarations and modules, are called "static" or "third-generation" languages. Examples of such languages are Pascal, C, and C++. Their main advantage is that they can be applied to large problems, and that they are efficient. Languages that avoid static constructs in order to obtain utmost flexibility with least effort are called "dynamic" or "fourth-generation" languages. Dynamic languages support incremental loading of code, garbage collection, and are closely tied to a supporting development environment. They excel at the rapid development of small pieces of throw-away code, e.g., scripts for component assembly. Their main virtue is that "everything goes", i.e., they don't have rigid type systems, verbose declaration sections, or similarly constraining static constructs. Even with the most aggressive compiler optimization techniques, they are typically slower than static languages. Close integration with the environment means that their use is convenient, and that an object model can be directly fitted to the language, so that interfacing between components becomes much easier as when using a language-independent object model.

Concerning component software, it is interesting to note that the gap between static and dynamic languages is what caused OLE and OpenDoc to end up with two levels of programmability: at the object model level (static) and at the automation level (dynamic).

More modern languages such as Component Pascal and Java have shown that the gap between static and

dynamic languages can be overcome, i.e., that a language can successfully bridge the gap by combining most advantages of both extremes. Such a hybrid language allows flexible development or modification of component implementations. On the other hand it allows to specify interfaces rigidly, so that conformance to these interfaces can be checked automatically. We call such a language, e.g., Java and Component Pascal, "component-oriented". So far we haven't talked about object-orientation, which appears both in static and in dynamic languages. So let's take a look at what object-orientation (OOP) is, and how component-orientation goes beyond OOP.

Not everyone agrees what OOP is or should be, but most people would require the following features in an OOP language: objects, classes, polymorphism, late binding, information hiding and inheritance.
An object encapsulates state and behavior. The behavior of an object is provided through procedures bound to the object's type, in so-called methods. A class is a blueprint for the implementation of a particular type of object. At run-time, an arbitrary number of instances, i.e., objects, of a class can be created.

Polymorphism means that sufficiently similar objects can be substituted for each other, i.e., a variable may be assigned objects of different types at run-time. Objects are sufficiently similar if they implement the same interface, i.e., if they fulfill the same contract. For example, a storage mechanism should accept any object which is storable, i.e., any object which implements the storage interface of the mechanism. A storage interface may contain the two procedures *Externalize* and *Internalize*, where *Externalize* causes the object's contents to be written to a file, while *Internalize* reads the object's contents from a file.

Late binding means that an object's behavior may be different, depending on which dynamic type it has. For example, some storable objects have different contents, and thus have different implementations of their *Externalize* procedures. A triangle object externalizes the coordinates of its three vertices, while a text object externalizes the character sequence that it contains. The name "late binding" stems from the fact that, due to polymorphism, it isn't known at compilation time whether a storable object will be a triangle, a text, or something else. Consequently, it must be decided later, namely at run-time, what type of object is being externalized and where its correct *Externalize* code can be found.

Information hiding means that an object distinguishes between its interface and its implementation. Interaction with the outside only occurs through the interface, the implementation is hidden. This allows to change hidden details of an implementation later, without invalidating the source code of clients.

Polymorphism, late binding and information hiding work together to make a clear separation of interface and implementation feasible, and thus support component software. However, implementation inheritance is something different altogether. It means that an object can "inherit" some behavior from another object, "override" part of it, and add its own new behavior. This is a convenient form of code reuse. It works fine if an inheriting object strictly adheres to the contract of the object from which it inherits, because then it can be substituted for it without breaking the contract with the clients.
Unfortunately, it is very difficult to prevent reentrant use of an object if inheritance is used, i.e., self-recursion can lead to unpredictable state changes in the inherited object. For example, the control flow within an object may jump up and down between baseclass and subclass in a complex "yo-yo" pattern. If some implementation detail of the base class is modified in a new release, the subclass may break because its assumptions are no longer valid.
Information hiding makes it impossible to specify the contract between subclass and baseclass unambiguously (the so-called specialization interface). Implementation inheritance is such a tight coupling between objects that in practice it requires source code of the inherited object to be made available, i.e., information hiding must be given up and the implementation must serve as interface ("inheritance breaks encapsulation"). We have met this problem earlier under the name "semantic fragile base class problem". In the future, someone may come up with practical rules for a restricted kind of implementation inheritance that does not provoke the semantic fragile base class problem. But today, this is still a research problem. The book [Szyperski97] goes into more details about this problem.

It may well turn out that implementation inheritance is the GOTO statement of the nineties. Like the GOTO in the sixties, inheritance is very convenient, programmers are used to it, it is not always obvious how to do without it, doing without it can make a program longer, and the issue can cause heated debates. But more fundamentally, inheritance resembles the GOTO in that it causes uncontrollable transfers of control, which makes it difficult to understand a program, and risky to modify it.

Inheritance is harmful if used across component boundaries. Whether inheritance is used inside of a component is irrelevant. Since a component is a black box, it may be implemented using implementation inheritance, functional programming, assembly language, or whatever is suitable for this particular component. The only thing that matters is that the interface is implemented correctly, i.e., that the contract with the outside world is not violated. Within a component, you have exclusive control over all your source code and may freely change internal interfaces whenever you want.

Component-oriented languages help to create more reliable component software systems faster; by providing "component-oriented" features in addition to the OOP features polymorphism, late binding, and information hiding.

Safety is such a feature. Safety means that a language guarantees some basic "rights" of a component, which need not be put down in every interface contract anew. In particular, a safe programming language guarantees memory integrity, i.e., one component cannot destroy another component's memory. This simplifies the contractual obligations of every component, since correct memory management - otherwise the source of about half of all programming errors - can simply be taken for granted. This is achieved by providing a garbage collection service, i.e., memory is reclaimed automatically when it isn't used anymore. Garbage collection is invisible, and relieves programmers of manual deallocation.
Safe languages yield the kind of protection that is desirable in a software environment consisting of tightly interacting components, especially in view of the fact that traditional hardware protection mechanisms are not applicable.

In the future, more and more customers could demand the use of safe languages for the construction of a component, because this can greatly reduce the number of mysterious crashes, and thus the distrust in the component. Once components become ubiquitous, quality issues necessarily become a major concern.

A component-oriented language also provides the means to achieve safety on a higher level than basic memory integrity only. Information hiding is part of the answer, since it allows to hide, and thus to protect, implementation details. Most OOP languages limit information hiding to single classes. This is too restrictive, since typically several classes must cooperate to provide a particular service. These classes must be able to work together closely, while their cooperation should be protected from outside interference. This means that these classes have their own private interface, to be shared with no one else. To guarantee compliance with this kind of private contract, a programming language must support information hiding across several classes. To do this, a language should provide a "module" or "package" construct, like the languages Component Pascal or Java.
Information hiding beyond single classes is an important requirement for a component-oriented language, because it enables a software architect to establish *customized safety properties* (i.e., invariants) throughout a component software system.

A component-oriented language implies that an implementation provides an object model which supports the dynamic loading of new components. Typically, there is a library service that allows to explicitly load a component, given its name or another suitable identifier. Such a facility is called metaprogramming support, since one program can manipulate (in this case load) another program. Reflection is another common term. Often, a library provides extensive run-time access to items of a loaded component. This requires extensive run-time type information (RTTI) that goes far beyond the minimal information supported by OOP languages

such as C++.
The following table gives a historical perspective to the evolution of imperative programming:

| Decade | Programming technology | Key advance |
|---|---|---|
| 1940s | machine codes | programmable machines |
| 1950s | assembly languages | symbols |
| 1960s | high-level languages | expressions and machine-independence |
| 1970s | structured programming | structured types and control structures |
| 1980s | modular programming | separation of interface from implementation |
| 1990s | object-oriented programming | polymorphism |
| 2000s | component-oriented programming | dynamic and safe composition |

Table 4-1. Evolution of imperative programming

Each step is an evolution from its predecessor and is characterized by one or two key advances. The step from machine code to assembler was based on the introduction of symbols, a step which has often been pivotal in other fields of human endeavour. High-level languages introduced machine-independence and expressions as exemplified by Fortran which is a shortening of "formula translation". Structured programming made two steps: definable types applied to data; and code, at every level of detail, was constrained to have the form of a "pipe" with one way in and one way out. Modular programming, as exemplified by Modula-2 and Ada, introduced the formality of an interface on each software component. Object-oriented programming added polymorphism, i.e., extensible types. Finally, the step which makes component software a reality, by enabling independently evolving components to cooperate, is the dynamic loading and safe integration of components.

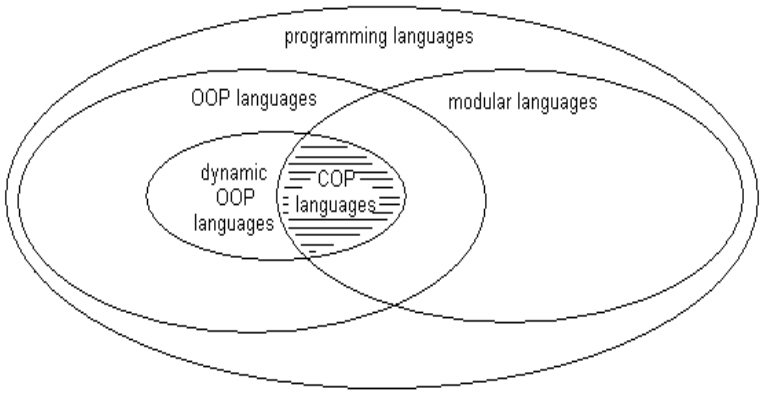


Figure 4-2. Component-oriented programming languages

We can now summarize the most important conclusions concerning component software and programming languages: the choice of a programming language can influence a project's cost during most of a component's life cycle; some languages support component software decidedly better than others; and standardized language-independent object models make the choice of language less strategic than it used to be.

In short: there is no good reason not to use the best language for the job.

Probably there will always be a choice of languages. On the one hand, existing languages never die; witness Fortran, Cobol, Basic, or Pascal. On the other hand, one-language approaches have never been successful. It was often thought that a language would be the last one; witness PL/1, Ada, C++, and now Java.
In fact, standardized object models could even make the development of new languages more attractive, if

(and only if) the language designers take care from the outset that they allow for good support of the important object models on the market.

Table 4-3 compares several programming languages with respect to their support for component software. It is not the intention to provide an exhaustive comparison of all existing languages; only the most important general-purpose languages have been considered.

| Aspect | Pascal | Modula-2 | C | C++ | Smalltalk | Java | Comp. Pascal |
|---|---|---|---|---|---|---|---|
| structured syntax | yes | yes | no | no | no | no | yes |
| simplicity and regularity | yes | yes | no | no | yes | no (1) | yes |
| static objects | yes | yes | yes | yes | no | no | yes |
| static types | yes | yes | yes | yes | no | yes | yes |
| dynamic types | no | no | no | yes | yes | yes | yes |
| efficiently translatable | yes | yes | yes | yes | no | yes (2) | yes |
| dynamic binding | no | yes (3) | yes (3) | yes | yes | yes | yes |
| information hiding | no | yes (4) | no | yes (5) | yes (5) | yes (4) | yes (4) |
| polymorphism | no | no | no | yes | yes | yes | yes |
| inheritance | no | no | no | yes | yes | yes | yes |
| multiple inheritance | no | no | no | yes | no | yes (6) | no |
| full type safety | no (7) | no (7) | no (7, 8) | no (9) | yes | yes | yes |
| garbage collection (10) | no | no | no | no | yes | yes | yes (11) |
| dynamic loading | no | no | no | no | yes | yes | yes |
| distinct interf. / implem. | no | no | no | no | no | yes (6) | no (12) |

(1) Language definition includes 34 classes and hundreds of methods; there are many exceptions of the rules; the language features interlock in a way that it is difficult to explain one feature without knowing the others.
(2) Absence of static types and value parameters incur a certain performance penalty.
(3) Procedure types / pointers to procedures.
(4) Modules / packages.
(5) Within single classes only, no invariants across classes can be guaranteed.
(6) Interfaces for (multiple) interface inheritance; classes for (single) implementation inheritance.
(7) Unsafe variant records; explicit dispose.
(8) Unsafe pointers.
(9) Inherits unsafe C features.
(10) Garbage collection is necessary to achieve full type safety.
(11) Produces less garbage than languages without static types, thus more efficient.
(12) By convention, implementation inheritance is rarely used, thus separation is not overly important.

Table 4-3. Comparison of programming languages

Note that even truly component-oriented languages like Java and Component Pascal are not perfect. For example, they don't yet support pre- and postconditions in an interface, as Eiffel [Meyer89] does. (Eiffel is not component-oriented. Its type system requires global analysis, which contradicts the idea of component software; it lacks a package construct; and its libraries are strongly coupled through implementation inheritance.) Generics that are compatible with separately compiled and loaded components would also be desirable in Java and Component Pascal, to increase the opportunities for code reuse. (C++ templates are not compatible with component software in the sense that a template cannot be compiled separately from its clients; each instantiation has to generate new code with all the resulting duplication of code.)

Java is highly portable due to its virtual machine. This is its greatest strength. It is also a weakness, because it makes interfacing to legacy software and to hardware difficult or impossible without falling back to other languages.
Component Pascal is a modern component-oriented language that complements Java, because it is much smaller and simpler, and not bound to a virtual machine. These factors are particularly important for embedded systems, such as robot control systems. For cost reasons, even high-performance embedded

systems have only limited amounts of memory, and it is often necessary to write new device drivers for such applications. The *Portos* [Portos] operating system is designed for embedded real-time applications that have to meet hard real-time constraints, and yet have to be extensible in a component-oriented way. Portos is completely implemented in Component Pascal, and supports applications written in Java. The two languages are close enough that they can even share the same asynchronous real-time garbage collector.
Of course, due to its relative simplicity and regularity, Component Pascal is also well suited for teaching programming, although it was not designed for teaching in particular. It was designed to support the development *and evolution* of complex component-based software systems.

The previous discussions have given us a deeper understanding of what a component is, by looking at programming issues and desirable language support. To conclude this section with some additional insights into the nature of components, let us go back to the definition of a component given earlier:

*A component is a unit of composition with a contractually specified interface and explicit context dependencies only. Components can be deployed independently of each other and are subject to composition by third parties.*

Now we can add some further observations and definitions to this short one. A component is a black box that provides some services, through a so-called export interface, and requires some services from other components, through a so-called import interface. The import interface embodies the component's context dependencies.
A component is a unit for packaging, deployment and loading. A component can be loaded, but there is at most one instance of it in memory anytime, i.e., it doesn't exist in many instances like objects usually do. A component is rarely copied; usually for deployment and distribution only.
A component typically consists of several classes and possibly some objects, e.g., class factories. A component has a configuration, i.e., a certain state when it is loaded. The component defines a default configuration. This is often achieved by packaging some data files, so-called resources, along with the component's code. These resources may be forms, strings, or other parameters; they are an integral part of the component. Some parts of a component's configuration may be changed at run-time, e.g., by replacing the component's configuration objects, such as class factories. A component should be considered immutable, i.e., permanent changes to its configuration are recorded outside of it, e.g., in preference files or in a system registry.
In summary, a component is an immutable entity that internally consists of classes, configuration objects, and resources. In Component Pascal, the smallest component is a module.

### 4.2 Frameworks

A component has an export interface. This interface may contain as little as one procedure through which a handle to a class factory can be obtained. The class factory in turn can be used to allocate new objects of the classes that the component implements.
But an export interface may also define interfaces for *other* components. Such an interface constitutes a standard to which these other components conform. Components that conform to one or several common standards may be interoperable, components that conform to no common standard are not interoperable.

For example, OLE defines a number of COM interfaces. COM components that implement objects with these interfaces can be used in compound documents. As a refinement of the common definition of an application framework in [Lewis95], the following definition is given:

*A component framework is a collection of component interfaces that embodies an abstract design for solutions to a family of related problems.*

A component framework is a collection of contracts, i.e., rules that specify how objects can work together. Some of these rules may be mere conventions. Other rules may be easy to follow because the framework provides suitable code along with the interfaces. For example, GUI frameworks provide default behavior for applications, windows, menus, and so on. If the default behavior is not replaced, it can be expected to follow the rules, e.g., to implement the correct user interface guidelines of the underlying platform. Still other rules may actually be enforced by the framework, e.g., drawing routines may prevent an application from drawing outside of its windows. Enforcement means that the framework provides some services only through safe code that guarantees the necessary invariants (e.g., the invariant "drawing always occurs inside of the application's windows"). The key to the enforcement of such invariants, which typically span several objects, is the information hiding across several classes, as discussed in the previous section.

In contrast to the older application frameworks, a component framework defines rules for independently developed and dynamically loadable components, rather than for classes that are linked together into a monolithic application. A component framework may provide interfaces, possibly along with some procedures. In contrast to most application frameworks, component frameworks are black-box frameworks, i.e., frameworks that can be used without access to their source code. Like a perfect contract, a pure black-box interface is an ideal that can only be approximated in practice.
Application frameworks that heavily rely on implementation inheritance are white-box frameworks that are published together with their source code. In contrast to application frameworks, a component framework need not be a class library. In fact, a component framework may even contain no code at all; it may just be a collection of interfaces. This is a difference to older interpretations of the term "framework", where the existence of actual code played a more important role. In component frameworks, actual code mostly exists for rule enforcement.

Years ago, Microsoft tried a "Windows everywhere" strategy, i.e., one operating system for every possible use. Today, Microsoft is pushing an "Active Platform" strategy. One interpretation of this shift is that they realized that the actual code (i.e., Windows) is not important; that only the collection of interfaces (i.e., the COM interfaces constituting the ActiveX Platform) matter, as long as there is *some* implementation of the these interfaces. Hardware designers have understood the importance of architecture versus implementation since the IBM 360 in the sixties; software designers are only now catching up to this insight.

A framework embodies architecture, i.e., the design of extension components and their interactions. Implementing an extension component according to the standard defined by a framework's interfaces is a form of reuse: design reuse. Since developing a new design is so much more difficult than implementing an existing design, design reuse is more important than mere code reuse. Creating a new design requires knowledge of an application domain, experience with other designs, the capability to distinguish essential issues from unimportant ones, and a flair for recognizing and inventing patterns. Since bad designs can become very expensive, only the most experienced developers should create frameworks. Other programmers should focus on good implementations of existing designs, i.e., develop components that implement given interfaces. Even more programmers will concentrate on component assembly.

By embodying design, frameworks have to some degree become competitors to Computer-Aided-Software-Engineering (CASE) tools. The advantage of frameworks compared to object-oriented design methods (OODM) is that the design is directly specified in a genuine programming language, and thus conformance of extension components can be checked by the compiler. However, provided that they become more flexible, CASE tools may still play an important role in the *documentation* of frameworks, and possibly in the automatic generation of some kinds of components.

It is well documented that framework design is an iterative process [Lewis95]. In each iteration, a new solution based on the framework is developed. Usually, this experience leads to some modification of the framework and the other already existing extension components. It is hoped that after several iterations, the framework becomes stable enough that new extension components can be developed without further

framework modifications. For non-trivial frameworks, this process can easily take several years. The current hectic production of new CORBA, ActiveX and Java frameworks is dangerous because their definition often seems to develop faster than the necessary experience. Moreover, standardization and certification issues even further complicate the already difficult development process.

The time needed to obtain good framework designs has far-reaching financial implications. In particular, in-house developers in large enterprises need to be aware that developing reusable software components is almost like developing products for the global market: it requires a potentially large up-front investment, professional documentation, a market calibration phase, (internal) marketing and sales, and post-sales support. This approach is too expensive if the return-on-investment is calculated on a per-project basis - instead of considering the longer-term savings, time-to-market reduction, lower risk of failure, and better evolvability of component software systems.

### 4.3 Development environments

Most of us who use software today want it to solve a problem, and not to be a problem itself. Thus we are interested in solutions, and not in programming. In this sense, programming is an undesirable activity, to be avoided wherever possible. Unfortunately, each kind of customization, and this includes component assembly, will require programming to some degree. Many tool vendors promise that you can customize existing software "without writing one line of code".
Don't believe them. Computers are so dumb that you have to tell them exactly what you want of them. Whether or not this is called programming, it still requires all the necessary skills, such as precision and explicitness.
For example, there are some component assembly tools that allow to specify behavior in a graphical way, or worse, in a mixture of graphical and textual ways. A picture is worth a thousand words. This is true for describing specific situations, but rarely for describing behavior or general rules. Graphical programming tools can be suitable under special circumstances, where problems are very limited in size and scope. Small finite state machines are a good example. For more complex problems, development will become more cumbersome - and maintenance much more cumbersome - than with a solid "real" programming language.

Ideally, a development tool for component software should be based on a component-oriented language. Tools cannot fully compensate for the weaknesses of a programming language. For example, a safe language prevents whole classes of errors, which is simpler and less expensive than forcing programmers to debug them later, using fancy and expensive debugging tools. Concerning debugging, the old rule holds: the sooner an error is caught, the less expensive its correction is. If a language has an expressive type system, its compiler helps you detect errors already at compile-time, before you deliver the component to your customers. Even if an error can only be detected at run-time, it is easier to trace back the earlier it is detected. Checking possible error conditions, such as violations of a procedure's preconditions, is called a defensive programming style. It is a desirable programming style for any kind of production software.

For component assembly, a development tool can be expected to provide the basic RAD (Rapid Application Development) facilities known from tools such as Microsoft's Visual Basic or Borland's Delphi. Basically, this is a collection of controls such as buttons, text fields, list boxes, etc., and a graphical editor for controls. Typically, the editor allows to interactively lay out controls in a form and save the layout in a file. This is a simple form of a compound user interface, with forms and controls as persistent objects. A tool should not generate source code out of a layout, since this makes later changes cumbersome and impossible without access to the development tool, and possibly even the program sources.
Source code generating "wizards" are a fashionable feature of many RAD tools. However, they should only relieve you of obvious and repetitive typing chores. They should not perform any magic that you don't understand; otherwise the tool becomes your master, instead of the other way around. In general, it can be said that the better a programming language is, the less need there is for source code generating tools,

because the language will allow to put the repetitive code into a library rather than regenerating it again and again with only minor variations.

Generating source code out of a form layout is one extreme. But beware of the other extreme as well. The other extreme is source code embedded in user interface elements. It should be possible to obtain an overview over the complete source code of your component or component assembly without accessing user interface elements, i.e., code should not be hidden behind buttons or other "rat holes". If bits and pieces of your code are spread over hundreds of rat holes, maintenance becomes an extremely messy business that limits the growth potential of the software. Growth potential is important, because successful component assemblies tend to adapt to new needs and thus to grow over time. Thus think twice before choosing simple scripting tools that can only perform component assembly. You may solve 80% of your immediate problem very quickly with them, but the remaining 20% may take longer than the first 80%. In large enterprises, it may be appropriate to have specialized groups for component construction and for component assembly, but this separation should be driven by organizational considerations, not by the tools.

Flexible interface and source code browsers and search facilities are useful to give you quick access to information about library components and your own components. A symbolic debugger should support a defensive programming style, in particular since you can only debug your own components. Components that you bought will usually be black boxes without source code, thus you should find errors in your own code as early as possible, before the behavior of other components can be affected.

A component-oriented programming language already supports the notion of black-box components, e.g., through a module or package construct. Without such a construct, packaging a component becomes difficult. Pure object-oriented environments, like Smalltalk, often suffer from this problem. In such systems, it can become extremely difficult to find all objects that you need to pack together into a component, because you have no way of knowing whether you missed some objects. The packing of components is one problem of such understructured environments, maintenance is another one. Considering that about 80% of all software investments today are spent for maintenance, there is good reason not to make it even worse in the future.

Rapid application development is important to keep up with the quickly changing business environment that is typical today. Sometimes, this means that the most appropriate approach is to produce "throwaway" code as quickly as possible, so that it can be used, written off, and replaced quickly. This is better than producing the perfect solution for a business that already doesn't exist in this form anymore. For example, quick-and-dirty extensions realized with implementation inheritance are perfectly ok if the software is scrapped before it needs to be revised or adapted to new needs.
But rapid development and controlled evolution can be compatible. With suitable documentation and refactoring aids, a component-oriented RAD tool can both support rapid development and "rapid maintenance".

Ideally, a development tool supports the whole life cycle of software. Today, there are no tools yet that can do this for component software, since traditional object-oriented design methods and CASE tools are based on the closed-world assumption of monolithic software. Tools suitable for component software would have to support black-box components for which no source code is available, interface management of components, run-time configuration management, and continuous incremental evolution.
This last point is important: the life cycle of a component software system can be much longer than the life cycle of any of its components. Successful component software never dies, it is just gradually extended, or refreshed by replacing outdated components. In this sense, component software can be regarded as a systematic way to deal with legacy issues.

Component assembly and even component construction are easy compared to the design of new component frameworks. Since the controlled evolution of a component framework depends so much on programming language support for guaranteeing integrity and evolvability, it would be too expensive not to

use a component-oriented language for such an endeavour.

A very simple quick check of the quality of a component development tool is whether the tool itself is built out of components, i.e., whether it takes its own medicine. Otherwise it indicates that the tool designers are not confident enough of its power.

### 4.4 BlackBox Component Builder


**FINALIST**

In the remainder of this book, the BlackBox Component Builder development tool is used. It is based on the language Component Pascal, a component-oriented refinement of Pascal. If you know Pascal or Modula-2, you will soon feel at home with Component Pascal, and like its increased convenience and safety, e.g., its garbage collector. For a description of the differences between Pascal and Component Pascal, see Appendix A and Appendix B.

The BlackBox Component Builder has been designed as a component development tool from the beginning. It provides the usual RAD facilities such as controls and a forms-based user interface editor. However, it is not limited to forms-based design only. Controls are views, and forms are just a special example of a view container. Any other container can be used as basis for compound user interfaces, e.g., a hypertext container. The BlackBox Component Framework (BCF) defines a general abstraction for containers, designed in a way that user interface details are hidden. On Windows, container views look and feel like ActiveX containers; on Mac OS, container views look and feel like OpenDoc containers. The Windows version transparently supports OLE, i.e., the user sees no difference whether a view is an ActiveX object or a BCF view. BCF is platform-independent, i.e., source code can be transferred between all supported platforms by mere recompilation. Currently there are versions for Windows 3.1/95/NT and for Mac OS 7.

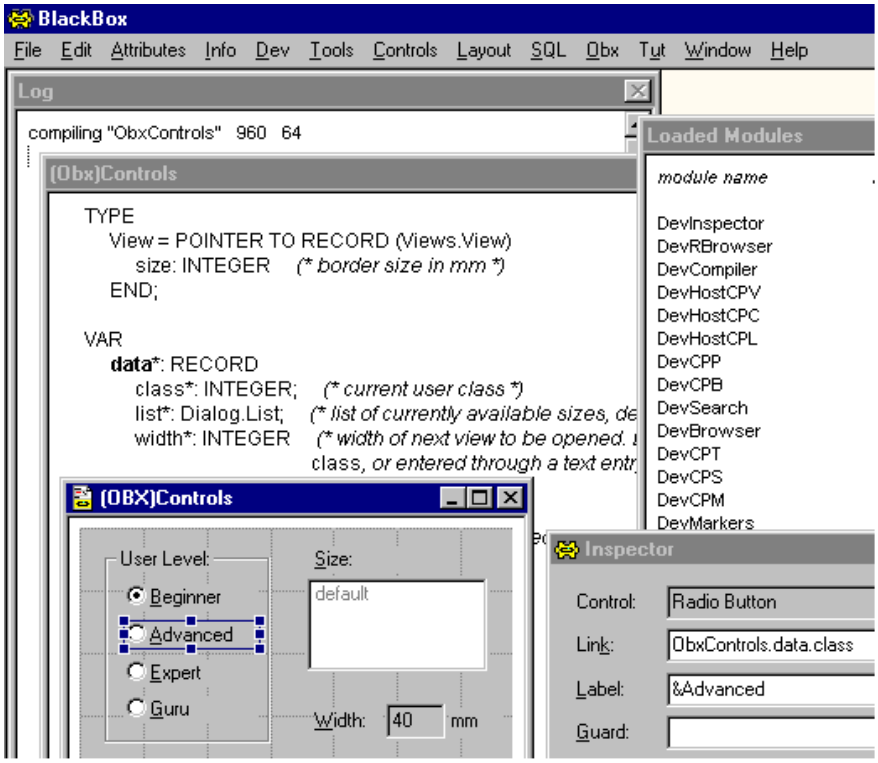

Figure 4-4. BlackBox Component Builder for Windows

Figure 4-5. BlackBox Component Builder for Macintosh

BCF has extensive support for compound documents and compound user interfaces. Consequently it is designed as a black-box framework, i.e., it uses implementation inheritance only rarely and only where it doesn't make interfaces ambiguous. Care has been taken to ensure a high degree of safety, e.g., to keep the effects of a malfunctioning view as local as possible.

One of the most important goals for BCF was to provide excellent maintainability. This is achieved by relying on composition rather than inheritance for code reuse, by clearly separating program logic from user interface code, and by making the architecture as explicit as possible, i.e., by expressing it in Component Pascal's powerful yet simple type system wherever feasible.

Although Component Pascal is a dynamic language with garbage collection, it is compiled straight into efficient machine code; no interpretation is involved. This gives good control over the run-time behavior of a program and allows to implement even computationally expensive algorithms, e.g., numerical algorithms. It is possible to use special compiler-supported libraries that permit low-level programming such as writing device drivers.
The BlackBox Component Builder is written 100% in Component Pascal; and the compiler, the visual designer, the hypertext subsystem, and so on are all implemented as Component Pascal modules, i.e., as components. Even the run-time system with its garbage collector has been written completely in Component Pascal.
A component consists of at least one module; components consisting of several modules are called subsystems. A module is Component Pascal's unit of compilation and loading, i.e., a large system can be compiled incrementally component by component, and is likewise loaded incrementally. Components that are not used are not loaded either. A programmer can explicitly cause new components to be loaded at run-time, in order to extend the system's capabilities while it runs. Although Component Pascal is compiled, the compiler is so fast and well integrated that it feels like an interpreted environment.
The BlackBox Component Builder allows to directly access non-Component-Pascal software. For example, the Windows versions supports the complete Windows NT APIs, access to arbitrary DLLs, and the creation of DLLs. A special version of the Component Pascal compiler even directly supports COM and DCOM ("Direct-To-COM Compiler with Safer OLE").
Figure 4-6 gives an architectural overview over the components constituting BCF. Rectangles with thin outlines are individual modules, while rectangles with thick outline are whole subsystems, i.e., collections of related modules:

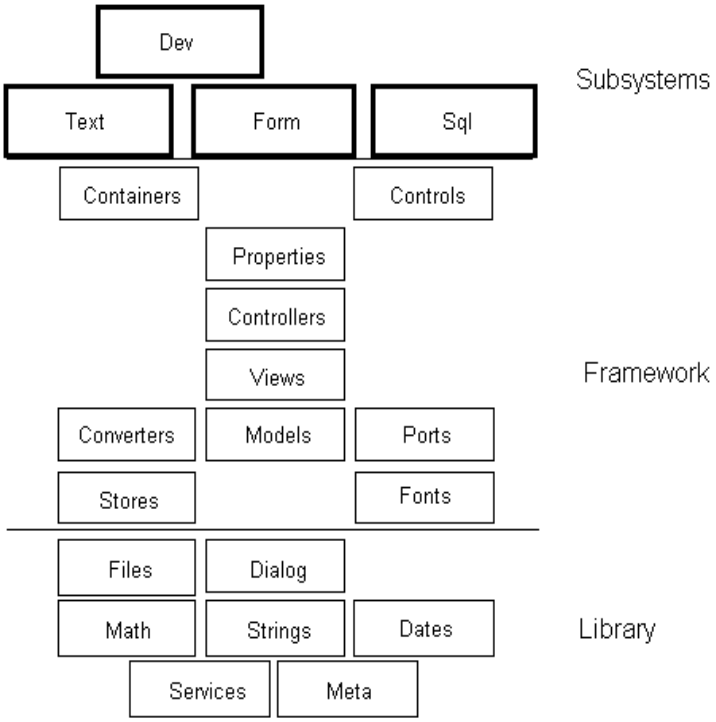


Figure 4-6. Architectural overview over the BlackBox Component Framework

At the lowest level, there is a library of largely independent modules providing a variety of low-level services, such as mathematical routines, metaprogramming support and files. On top of this base library, there is a number of modules which support a flexible and platform-independent compound document / compound user interface architecture. Built on top of this architecture, a text processing component, a forms editing component, and a development environment component are provided. Thanks to intensive reuse, this comprehensive system is so small that it fits on three floppy disks. For example, the symbolic debugger that is part of the Dev subsystem uses the Text subsystem to provide a convenient hypertext-like front-end to the symbolic debugger.
We will meet the most important modules of the framework in parts III and IV.