23

Development and Design

There is no silver bullet.
– F. Brooks

Building software — aims and means — development process — development cycle — design aims — design steps — finding classes — specifying operations — specifying dependencies — specifying interfaces — reorganizing class hierarchies — models — experimentation and analysis — testing — software maintenance — efficiency — management — reuse — scale — the importance of individuals — hybrid design — bibliography — advice.

23.1 Overview [design.overview]

This chapter is the first of three that present the production of software in increasing detail, starting from a relatively high-level view of design and ending with C++ specific programming techniques and concepts directly supporting such design. After the introduction and a brief discussion of the aims and means of software development in §23.3, this chapter has two major parts:

§23.4 A view of the software development process

§23.5 Practical observations about the organization of software development

Chapter 24 discusses the relationship between design and programming language. Chapter 25 presents some roles that classes play in the organization of software from a design perspective. Taken as a whole, the three chapters of Part 4 aim to bridge the gap between would-be language-independent design and programming that is myopically focussed on details. Both ends of this spectrum have their place in a large project, but to avoid disaster and excessive cost, they must be part of a continuum of concerns and techniques.

23.2 Introduction [design.intro]

Constructing any nontrivial piece of software is a complex and often daunting task. Even for an individual programmer, the actual writing of program statements is only one part of the process. Typically, issues of problem analysis, overall program design, documentation, testing, and maintenance, as well as the management of all of this, dwarf the task of writing and debugging individual pieces of code. Naturally, one might simply label the totality of these activities ''programming'' and thereafter make a logically coherent claim that ''I don't design, I just program;'' but whatever one calls the activity, it is important sometimes to focus on its individual parts – just as it is important occasionally to consider the complete process. Neither the details nor the big picture must be permanently lost in the rush to get a system shipped – although often enough that is exactly what happens.

This chapter focusses on the parts of program development that do not involve writing and debugging individual pieces of code. The discussion is less precise and less detailed than the discussions of individual language features and specific programming techniques presented elsewhere in this book. This is necessary because there can be no cookbook method for creating good software. Detailed ''how to'' descriptions can exist for specific well-understood kinds of applications, but not for more general application areas. There is no substitute for intelligence, experience, and taste in programming. In consequence, this chapter offers only general advice, alternative approaches, and cautionary observations.

The discussion is hampered by the abstract nature of software and the fact that techniques that work for smaller projects (say, for one or two people writing 10,000 lines of code) do not necessarily scale to medium and large projects. For this reason, some discussions are formulated in terms of analogies from less abstract engineering disciplines rather than in terms of code examples. Please remember that ''proof by analogy'' is fraud, so analogy is used here for exposition only. Discussions of design issues phrased in C++ specific terms and with examples can be found in Chapter 24 and Chapter 25. The ideas expressed in this chapter are reflected in both the C++ language itself and in the presentation of the individual examples throughout this book.

Please also remember that because of the extraordinary diversity of application areas, people, and program-development environments, you cannot expect every observation made here to apply directly to your current problem. The observations are drawn from real-life projects and apply to a wide variety of situations, but they cannot be considered universal. Look at these observations with a healthy degree of skepticism.

C++ can be used simply as a better C. However, doing so leaves the most powerful techniques and language features unused so that only a small fraction of the potential benefits of using C++ will be gained. This chapter focusses on approaches to design that enable effective use of C++'s data abstraction and object-oriented programming facilities; such techniques are often called *object-oriented design*.

A few major themes run through this chapter:

- The most important single aspect of software development is to be clear about what you are trying to build.
- Successful software development is a long-term activity.
- The systems we construct tend to be at the limit of the complexity that we and our tools can handle.

- There are no ‘‘cookbook’’ methods that can replace intelligence, experience, and good taste in design and programming.
- Experimentation is essential for all nontrivial software development.
- Design and programming are iterative activities.
- The different phases of a software project, such as design, programming, and testing, cannot be strictly separated.
- Programming and design cannot be considered without also considering the management of these activities.

It is easy – and typically expensive – to underestimate any of these points. It is hard to transform the abstract ideas they embody into practice. The need for experience should be noted. Like boat building, bicycling, and programming, design is not a skill that can be mastered through theoretical study alone.

Too often, we forget the human aspects of system building and consider the software development process as simply ‘‘a series of well-defined steps, each performing specific actions on inputs according to predefined rules to produce the desired outputs.’’ The very language used conceals the human involvement! Design and programming are human activities; forget that and all is lost.

This chapter is concerned with the design of systems that are ambitious relative to the experience and resources of the people building the system. It seems to be the nature of individuals and organizations to attempt projects that are at the limits of their ability. Projects that don’t offer such challenges don’t need a discussion of design. Such projects already have established frameworks that need not be upset. Only when something ambitious is attempted is there a need to adopt new and better tools and procedures. There is also a tendency to assign projects that ‘‘we know how to do’’ to relative novices who don’t.

There is no ‘‘one right way’’ to design and build all systems. I would consider belief in ‘‘the one right way’’ a childhood disease, if experienced programmers and designers didn’t succumb to it so often. Please remember that just because a technique worked for you last year and for one project, it does not follow that it will work unmodified for someone else or for a different project. It is most important to keep an open mind.

Clearly, much of the discussion here relates to larger-scale software development. Readers who are not involved in such development can sit back and enjoy a look at the horrors they have escaped. Alternatively, they can look for the subset of the discussion that relates to individual work. There is no lower limit to the size of programs for which it is sensible to design before starting to code. There is, however, a lower limit for which any particular approach to design and documentation is appropriate. See §23.5.2 for a discussion of issues of scale.

The most fundamental problem in software development is complexity. There is only one basic way of dealing with complexity: divide and conquer. A problem that can be separated into two sub-problems that can be handled separately is more than half solved by that separation. This simple principle can be applied in an amazing variety of ways. In particular, the use of a module or a class in the design of systems separates the program into two parts – the implementation and its users – connected only by an (ideally) well-defined interface. This is the fundamental approach to handling the inherent complexity of a program. Similarly, the process of designing a program can be broken into distinct activities with (ideally) well-defined interactions between the people involved. This is the basic approach to handling the inherent complexity of the development process and the people involved in it.

In both cases, the selection of the parts and the specification of the interfaces between the parts is where the most experience and taste is required. Such selection is not a simple mechanical process but typically requires insights that can be achieved only through a thorough understanding of a system at suitable levels of abstraction (see §23.4.2, §24.3.1, and §25.3). A myopic view of a program or of a software development process often leads to seriously flawed systems. Note also that for both people and programs, *separation* is easy. The hard part is to ensure effective *communication* between parties on different sides of a barrier without destroying the barrier or stifling the communication necessary to achieve cooperation.

This chapter presents an approach to design, not a complete design method. A complete formal design method is beyond the scope of this book. The approach presented here can be used with different degrees of formalization and as the basis for different formalizations. Similarly, this chapter is not a literature survey and does not attempt to touch every topic relevant to software development or to present every viewpoint. Again, that is beyond the scope of this book. A literature survey can be found in [Booch,1994]. Note that terms are used here in fairly general and conventional ways. Most ''interesting'' terms, such as *design*, *prototype*, and *programmer*, have several different and often conflicting definitions in the literature. Please be careful not to read something unintended into what is said here based on specialized or locally precise definitions of the terms.

23.3 Aims and Means [design.aims]

The purpose of professional programming is to deliver a product that satisfies its users. The primary means of doing so is to produce software with a clean internal structure and to grow a group of designers and programmers skilled enough and motivated enough to respond quickly and effectively to change and opportunities.

Why? The internal structure of the program and the process by which it was created are ideally of no concern to the end user. Stronger: if the end user has to worry about how the program was written, then there is something wrong with that program. Given that, what is the importance of the structure of a program and of the people who create the program?

A program needs a clean internal structure to ease:

- testing,
- porting,
- maintenance,
- extension,
- reorganization, and
- understanding.

The main point is that every successful major piece of software has an extended life in which it is worked on by a succession of programmers and designers, ported to new hardware, adapted to unanticipated uses, and repeatedly reorganized. Throughout the software's life, new versions of it must be produced with acceptable error rates and on time. Not planning for this is planning to fail.

Note that even though end users ideally don't have to know the internal structure of a system, they might actually want to. For example, a user might want to know the design of a system in detail to be able to assess its likely reliability and potential for revision and extension. If the software in question is not a complete system – rather, a set of libraries for building other software –

then the users will want to know more ''details'' to be able to better use the libraries and also to better benefit from them as sources of ideas.

A balance has to be struck between the lack of an overall design for a piece of software and overemphasis on structure. The former leads to endless cutting of corners (''we'll just ship this one and fix the problem in the next release''). The latter leads to overelaborate designs in which essentials are lost in formalism and to situations where implementation gets delayed by program reorganizations (''but this new structure is *much* better than the old one; people will want to wait for it''). It also often results in systems so demanding of resources that they are unaffordable to most potential users. Such balancing acts are the most difficult aspects of design and the area in which talent and experience show themselves. The choices are hard for the individual designer or programmer and harder for the larger projects in which more people with differing skills are involved.

A program needs to be produced and maintained by an organization that can do this despite changes of personnel, direction, and management structure. A popular approach to coping with this problem has been to try to reduce system development into a few relatively low-level tasks slotted into a rigid framework. That is, the idea is to create a class of easy-to-train (cheap) and interchangeable low-level programmers (''coders'') and a class of somewhat less cheap but equally interchangeable (and therefore equally dispensable) designers. The coders are not supposed to make design decisions, while the designers are not supposed to concern themselves with the grubby details of coding. This approach often fails. Where it does work, it produces overly large systems with poor performance.

The problems with this approach are:

– insufficient communication between implementers and designers, which leads to missed opportunities, delays, inefficiencies, and repeated problems due to failure to learn from experience; and
– insufficient scope for initiative among implementers, which leads to lack of professional growth, lack of initiative, sloppiness, and high turnover.

Basically, such a system lacks feedback mechanisms to allow people to benefit from other people's experience. It is wasteful of scarce human talent. Creating a framework within which people can utilize diverse talents, develop new skills, contribute ideas, and enjoy themselves is not just the only decent thing to do but also makes practical and economic sense.

On the other hand, a system cannot be built, documented, and maintained indefinitely without some form of formal structure. Simply finding the best people and letting them attack the problem as they think best is often a good start for a project requiring innovation. However, as the project progresses, more scheduling, specialization, and formalized communication between the people involved in the project become necessary. By ''formal'' I don't mean a mathematical or mechanically verifiable notation (although that is nice, where available and applicable) but rather a set of guidelines for notation, naming, documentation, testing, etc. Again, a balance and a sense of appropriateness is necessary. A too-rigid system can prevent growth and stifle innovation. In this case, it is the manager's talent and experience that is tested. For the individual, the equivalent dilemma is to choose where to try to be clever and where to simply ''do it by the book.''

The recommendation is to plan not just for the next release of the current project but also for the longer term. Looking only to the next release is planning to fail. We must develop organizations and software development strategies aimed at producing and maintaining many releases of many projects; that is, we must plan for a series of successes.

The purpose of ''design'' is to create a clean and relatively simple internal structure, sometimes also called an *architecture*, for a program. In other words, we want to create a framework into which the individual pieces of code can fit and thereby guide the writing of those individual pieces of code.

A design is the end product of the design process (as far as there is an *end* product of an iterative process). It is the focus of the communication between the designer and the programmer and between programmers. It is important to have a sense of proportion here. If I – as an individual programmer – design a small program that I'm going to implement tomorrow, the appropriate level of precision and detail may be some scribbles on the back of an envelope. At the other extreme, the development of a system involving hundreds of designers and programmers may require books of specifications carefully written using formal or semi-formal notations. Determining a suitable level of detail, precision, and formality for a design is in itself a challenging technical and managerial task.

In this and the following chapters, I assume that the design of a system is expressed as a set of class declarations (typically with their private declarations omitted as spurious details) and their relationships. This is a simplification. Many more issues enter into a specific design; for example, concurrency, management of namespaces, uses of nonmember function and data, parameterization of classes and functions, organization of code to minimize recompilation, persistence, and use of multiple computers. However, simplification is necessary for a discussion at this level of detail, and classes are the proper focus of design in the context of C++. Some of these other issues are mentioned in passing in this chapter, and some that directly affect the design of C++ programs are discussed in Chapter 24 and Chapter 25. For a more detailed discussion and examples of a specific object-oriented design method, see [Booch,1994].

I leave the distinction between analysis and design vague because a discussion of this issue is beyond the scope of this book and is sensitive to variations in specific design methods. It is essential to pick an analysis method to match the design method and to pick a design method to match the programming style and language used.

23.4 The Development Process [design.process]

Software development is an iterative and incremental process. Each stage of the process is revisited repeatedly during the development, and each visit refines the end products of that stage. In general, the process has no beginning and no end. When designing and implementing a system, you start from a base of other people's designs, libraries, and application software. When you finish, you leave a body of design and code for others to refine, revise, extend, and port. Naturally, a specific project can have a definite beginning and end, and it is important (though often surprisingly hard) to delimit the project cleanly and precisely in time and scope. However, pretending that you are starting from a clean slate can cause serious problems. Pretending that the world ends at the ''final delivery'' can cause equally serious problems for your successors (often yourself in a different role).

One implication of this is that the following sections could be read in any order because the aspects of design and implementation can be almost arbitrarily interleaved in a real project. That is, ''design'' is almost always redesign based on a previous design and some implementation

experience. Furthermore, the design is constrained by schedules, the skills of the people involved, compatibility issues, etc. A major challenge to a designer/manager/programmer is to create order in this process without stifling innovation and destroying the feedback loops that are necessary for successful development.

The development process has three stages:

- Analysis: defining the scope of the problem to be solved
- Design: creating an overall structure for a system
- Implementation: writing and testing the code

Please remember the iterative nature of this process – it is significant that these stages are not numbered. Note that some major aspects of program development don't appear as separate stages because they ought to permeate the process:

- Experimentation
- Testing
- Analysis of the design and the implementation
- Documentation
- Management

Software ''maintenance'' is simply more iterations through this development process (§23.4.6).

It is most important that analysis, design, and implementation don't become too detached from each other and that the people involved share a culture so that they can communicate effectively. In larger projects, this is all too often not the case. Ideally, individuals move from one stage to another during a project; the best way to transfer subtle information is in a person's head. Unfortunately, organizations often establish barriers against such transfers, for example, by giving designers higher status and/or higher pay than ''mere programmers.'' If it is not practical for people to move around to learn and teach, they should at least be encouraged to talk regularly with individuals involved in ''the other'' stages of the development.

For small-to-medium projects, there often is no distinction made between analysis and design; these two phases have been merged into one. Similarly, in small projects there often is no distinction made between design and programming. Naturally, this solves the communication problems. It is important to apply an appropriate degree of formality for a given project and to maintain an appropriate degree of separation between these phases (§23.5.2). There is no one right way to do this.

The model of software development described here differs radically from the traditional ''waterfall model.'' In a waterfall model, the development progresses in an orderly and linear fashion through the development stages from analysis to testing. The waterfall model suffers from the fundamental problem that information tends to flow only one way. When problems are found ''downstream,'' there is often strong methodological and organizational pressure to provide a local fix; that is, there is pressure to solve the problem without affecting the previous stages of the process. This lack of feedback leads to deficient designs, and the local fixes lead to contorted implementations. In the inevitable cases in which information does flow back toward the source and cause changes to the design, the result is a slow and cumbersome ripple effect through a system that is geared to prevent the need for such change and therefore unwilling and slow to respond. The argument for ''no change'' or for a ''local fix'' thus becomes an argument that one suborganization cannot impose large amounts of work on other suborganizations ''for its own convenience.'' In particular, by the time a major flaw is found there has often been so much paperwork generated

relating to the flawed decision that the effort involved in modifying the documentation dwarfs the effort needed to fix the code. In this way, paperwork can become the major problem of software development. Naturally, such problems can – and do – occur however one organizes the development of large systems. After all, *some* paperwork is essential. However, the pretense of a linear model of development (a waterfall) greatly increases the likelihood that this problem will get out of hand.

The problem with the waterfall model is insufficient feedback and the inability to respond to change. The danger of the iterative approach outlined here is a temptation to substitute a series of nonconverging changes for real thought and progress. Both problems are easier to diagnose than to solve, and however one organizes a task, it is easy and tempting to mistake activity for progress. Naturally, the emphasis on the different stages of the development process changes as a project progresses. Initially, the emphasis is on analysis and design, and programming issues receive less attention. As time passes, resources shift towards design and programming and then become more focussed on programming and testing. However, the key is never to focus on one part of the analysis/design/implementation spectrum to the exclusion of all other concerns.

Remember that no amount of attention to detail, no application of proper management technique, no amount of advanced technology can help you if you don't have a clear idea of what you are trying to achieve. More projects fail for lack of well-defined and realistic goals than for any other reason. Whatever you do and however you go about it, be clear about your aims, define tangible goals and milestones, and don't look for technological solutions to sociological problems. On the other hand, do use whatever *appropriate* technology is available – even if it involves an investment; people do work better with appropriate tools and in reasonable surroundings. Don't get fooled into believing that following this advice is easy.

23.4.1 The Development Cycle [design.cycle]

Developing a system should be an iterative activity. The main loop consists of repeated trips through this sequence:

[0] Examine the problem.
[1] Create an overall design.
[2] Find standard components.
  – Customize the components for this design.
[3] Create new standard components.
  – Customize the components for this design.
[4] Assemble the design.

As an analogy, consider a car factory. For a project to start, there needs to be an overall design for a new type of car. This first cut will be based on some kind of analysis and specifies the car in general terms related mostly to its intended use rather than to details of how to achieve desired properties. Deciding which properties are desirable – or even better, providing a relatively simple guide to deciding which properties are desirable – is often the hardest part of a project. When done well, this is typically the work of a single insightful individual and is often called a *vision*. It is quite common for projects to lack such clear goals – and for projects to falter or fail for that reason.

Say we want to build a medium-sized car with four doors and a fairly powerful engine. The first stage in the design is most definitely not to start designing the car (and all of its sub-

components) from scratch. A software designer or programmer in a similar circumstance might unwisely try exactly that.

The first stage is to consider which components are available from the factory's own inventory and from reliable suppliers. The components thus found need not be exactly right for the new car. There will be ways of customizing the components. It might even be possible to affect the specification of the ''next release'' of such components to make them more suitable for our project. For example, there may be an engine available with the right properties except for a slight deficiency in delivered power. Either we or the engine supplier might be able to add a turbocharger to compensate without affecting the basic design. Note that making such a change ''without affecting the basic design'' is unlikely unless the original design anticipated at least some form of customization. Such customization will typically require cooperation between you and your engine supplier. A software designer or programmer has similar options. In particular, polymorphic classes and templates can often be used effectively for customization. However, don't expect to be able to effect arbitrary extensions without foresight by or cooperation with the provider of such a class.

Having run out of suitable standard components, the car designer doesn't rush to design optimal new components for the new car. That would simply be too expensive. Assume that there were no suitable air conditioning unit available and that there was a suitable L-shaped space available in the engine compartment. One solution would be to design an L-shaped air conditioning unit. However, the probability that this oddity could be used in other car types – even after extensive customization – is low. This implies that our car designer will not be able to share the cost of producing such units with the designers of other car types and that the useful life of the unit will be short. It will thus be worthwhile to design a unit that has a wider appeal; that is, design a unit that has a cleaner design and is more suited for customization than our hypothetical L-shaped oddity. This will probably involve more work than the L-shaped unit and might even involve a modification of the overall design of our car to accommodate the more general-purpose unit. Because the new unit was designed to be more widely useful than our L-shaped wonder, it will presumably need a bit of customization to fit our revised needs perfectly. Again, the software designer or programmer has a similar option. That is, rather than writing project-specific code the designer can design a new component of a generality that makes it a good candidate to become a standard in some universe.

Finally, when we have run out of potential standard components we assemble the ''final'' design. We use as few specially designed widgets as possible because next year we will have to go through a variant of this exercise again for the next new model and the specially designed widgets will be the ones we most likely will have to redo or throw away. Sadly, the experience with traditionally designed software is that few parts of a system can even be recognized as discrete components, and few of those are of use outside their original project.

I'm not saying that all car designers are as rational as I have outlined in this analogy or that all software designers make the mistakes mentioned. On the contrary, this model can be made to work with software. In particular, this chapter and the next present techniques for making it work with C++. I do claim, however, that the intangible nature of software makes those mistakes harder to avoid (§24.3.1, §24.3.4), and in §23.5.3 I argue that corporate culture often discourages people from using the model outlined here.

Note that this model of development really works well only when you consider the longer term. If your horizon extends only to the next release, the creation and maintenance of standard components makes no sense. It will simply be seen as spurious overhead. This model is suggested for an

organization with a life that spans several projects and of a size that makes worthwhile the necessary extra investment in tools (for design, programming, and project management) and education (of designers, programmers, and managers). It is a sketch of a kind of software factory. Curiously enough, it differs only in scale from the practices of the best individual programmers, who over the years build up a stock of techniques, designs, tools, and libraries to enhance their personal effectiveness. It seems, in fact, that most organizations have failed to take advantage of the best personal practices due to both a lack of vision and an inability to manage such practices on more than a very small scale.

Note that it is unreasonable to expect ''standard components'' to be universally standard. There will exist a few international standard libraries. However, most components will be standard (only) within a country, an industry, a company, a product line, a department, an application area, etc. The world is simply too large for universal standards to be a realistic or indeed a to be desirable aim for all components and tools.

Aiming for universality in an initial design is a prescription for a project that will never be completed. One reason that the development cycle is a cycle is that it is essential to have a working system from which to gain experience (§23.4.3.6).

23.4.2 Design Aims [design.design]

What are the overall aims of a design? Simplicity is one, of course, but simplicity according to what criteria? We assume that a design will have to evolve. That is, the system will have to be extended, ported, tuned, and generally changed in a number of ways that cannot all be foreseen. Consequently, we must aim for a design and an implemented system that is simple under the constraint that it will be changed in many ways. In fact, it is realistic to assume that the requirements for the system will change several times between the time of the initial design and the first release of the system.

The implication is that the system must be designed to *remain* as simple as possible under a sequence of changes. We must design for change; that is, we must aim for

- flexibility,
- extensibility, and
- portability.

This is best done by trying to encapsulate the areas of a system that are likely to change and by providing non-intrusive ways for a later designer/programmer to modify the behavior of the code. This is done by identifying the key concepts of an application and giving each class the exclusive responsibility for the maintenance of all information relating to a single concept. In that case, a change can be effected by a modification of that class only. Ideally, a change to a single concept can be done by deriving a new class (§23.4.3.5) or by passing a different argument to a template. Naturally, this ideal is much easier to state than to follow.

Consider an example. In a simulation involving meteorological phenomena, we want to display a rain cloud. How do we do that? We cannot have a general routine to display the cloud because what a cloud looks like depends on the internal state of the cloud, and that state should be the sole responsibility of the cloud.

A first solution to this problem is to let the cloud display itself. This style of solution is acceptable in many limited contexts. However, it is not general because there are many ways to view a

cloud: for example, as a detailed picture, as a rough outline, or as an icon on a map. In other words, what a cloud looks like depends on both the cloud and its environment.

A second solution to the problem is to make the cloud aware of its environment and then let the cloud display itself. This solution is acceptable in even more contexts. However, it is still not a general solution. Having the cloud know about such details of its environment violates the dictum that a class is responsible for one thing only and that every ''thing'' is the responsibility of some class. It may not be possible to come up with a coherent notion of ''the cloud's environment'' because in general what a cloud looks like depends on both the cloud and the viewer. Even in real life, what the cloud looks like to me depends rather strongly on how I look at it; for example, with my naked eyes, through a polarizing filter, or with a weather radar. In addition to the viewer and the cloud, some ''general background'' such as the relative position of the sun might have to be taken into account. Adding other objects, such as other clouds and airplanes, further complicates the matter. To make life really hard for the designer, add the possibility of having several simultaneous viewers.

A third solution is to have the cloud – and other objects such as airplanes and the sun – describe themselves to a viewer. This solution has sufficient generality to serve most purposes†. It may, however, impose a significant cost in both complexity and run-time overhead. For example, how do we arrange for a viewer to understand the descriptions produced by clouds and other objects?

Rain clouds are not particularly common in programs (but for an example, see §15.2), but objects that need to be involved in a variety of I/O operations are. This makes the cloud example relevant to programs in general and to the design of libraries in particular. C++ code for a logically similar example can be found in the manipulators used for formatted output in the stream I/O system (§21.4.6, §21.4.6.3). Note that the third solution is not ''the right solution;'' it is simply the most general solution. A designer must balance the various needs of a system to choose the level of generality and abstraction that is appropriate for a given problem in a given system. As a rule of thumb, the right level of abstraction for a long-lived program is the most general you can comprehend and afford, *not* the absolutely most general. Generalization beyond the scope of a given project and beyond the experience of the people involved can be harmful; that is, it can cause delays, unacceptable inefficiencies, unmanageable designs, and plain failure.

To make such techniques manageable and economical, we must also design and manage for reuse (§23.5.1) and not completely forget about efficiency (§23.4.7).

23.4.3 Design Steps [design.steps]

Consider designing a single class. Typically, this is *not* a good idea. Concepts do *not* exist in isolation; rather, a concept is defined in the context of other concepts. Similarly, a class does not exist in isolation but is defined together with logically related classes. Typically, one works on a set of related classes. Such a set is often called a *class library* or a *component*. Sometimes all classes in a component constitute a single class hierarchy, sometimes they are members of a single namespace, and sometimes they are a more ad-hoc collection of declarations (§24.4).

† Even this model is unlikely to be sufficient for extreme cases like high-quality graphics based on ray tracing. I suspect that achieving such detail requires the designer to move to a different level of abstraction.

The set of classes in a component is united by some logical criteria, often by a common style and often by a reliance on common services. A component is thus the unit of design, documentation, ownership, and often reuse. This does not mean that if you use one class from a component, you must understand and use all the classes from the component or maybe get the code for every class in the component loaded into your program. On the contrary, we typically strive to ensure that a class can be used with only minimal overhead in machine resources and human effort. However, to use any part of a component we need to understand the logical criteria that define the component (hopefully made abundantly clear in the documentation), the conventions and style embodied in the design of the component and its documentation, and the common services (if any).

So consider how one might approach the design of a component. Because this is often a challenging task, it is worthwhile breaking it into steps to help focus on the various subtasks in a logical and complete way. As usual, there is no one right way of doing this. However, here is a series of steps that have worked for some people:

[1] Find the concepts/classes and their most fundamental relationships.
[2] Refine the classes by specifying the sets of operations on them.
  – Classify these operations. In particular, consider the needs for construction, copying, and destruction.
  – Consider minimalism, completeness, and convenience.
[3] Refine the classes by specifying their dependencies.
  – Consider parameterization, inheritance, and use dependencies.
[4] Specify the interfaces.
  – Separate functions into public and protected operations.
  – Specify the exact type of the operations on the classes.

Note that these are steps in an iterative process. Typically, several loops through this sequence are needed to produce a design one can comfortably use for an initial implementation or a re-implementation. One advantage of well-done analysis and data abstraction as described here is that it becomes relatively easy to reshuffle class relationships even after code has been written. This is never a trivial task, though.

After that, we implement the classes and go back and review the design based on what was learned from implementing them. In the following subsections, I discuss these steps one by one.

23.4.3.1 Step 1: Find Classes [design.find]

Find the concepts/classes and their most fundamental relationships. The key to a good design is to model some aspect of ''reality'' directly – that is, capture the concepts of an application as classes, represent the relationships between classes in well-defined ways such as inheritance, and do this repeatedly at different levels of abstraction. But how do we go about finding those concepts? What is a practical approach to deciding which classes we need?

The best place to start looking is in the application itself, as opposed to looking in the computer scientist's bag of abstractions and concepts. Listen to someone who will become an expert user of the system once it has been built and to someone who is a somewhat dissatisfied user of the system being replaced. Note the vocabulary they use.

It is often said that the nouns will correspond to the classes and objects needed in the program; often that is indeed the case. However, that is by no means the end of the story. Verbs may denote

operations on objects, traditional (global) functions that produce new values based on the value of their arguments, or even classes. As examples of the latter, note the function objects (§18.4) and manipulators (§21.4.6). Verbs such as ''iterate'' or ''commit'' can be represented by an iterator object and an object representing a database commit operation, respectively. Even adjectives can often usefully be represented by classes. Consider the adjectives ''storable,'' ''concurrent,'' ''registered,'' and ''bounded.'' These may be classes intended to allow a designer or programmer to pick and choose among desirable attributes for later-designed classes by specifying virtual base classes (§15.2.4).

Not all classes correspond to application-level concepts. For example, some represent system resources and implementation-level abstractions (§24.3.1). It is also important to avoid modeling an old system too closely. For example, we don't want a system that is centered around a database to faithfully replicate aspects of a manual system that exist only to allow individuals to manage the physical shuffling of pieces of paper.

Inheritance is used to represent commonality among concepts. Most important, it is used to represent hierachical organization based on the behavior of classes representing individual concepts (§1.7, §12.2.6, §24.3.2). This is sometimes referred to as *classification* or even *taxonomy*. Commonality must be actively sought. Generalization and classification are high-level activities that require insight to give useful and lasting results. A common base should represent a more general concept rather than simply a similar concept that happens to require less data to represent.

Note that the classification should be of aspects of the concepts that we model in our system, rather than aspects that may be valid in other areas. For example, in mathematics a circle is a kind of an ellipse, but in most programs a circle should not be derived from an ellipse or an ellipse derived from a circle. The often-heard arguments ''because that's the way it is in mathematics'' and ''because the representation of a circle is a subset of that of an ellipse'' are not conclusive and most often wrong. This is because for most programs, the key property of a circle is that it has a center and a fixed distance to its perimeter. All behavior of a circle (all operations) must maintain this property (invariant; §24.3.7.1). On the other hand, an ellipse is characterized by two focal points that in many programs can be changed independently of each other. If those focal points coincide, the ellipse looks like a circle, but it is not a circle because its operations do not preserve the circle invariant. In most systems, this difference will be reflected by having a circle and an ellipse provide sets of operations that are not subsets of each other.

We don't just think up a set of classes and relationships between classes and use them for the final system. Instead, we create an initial set of classes and relationships. These are then refined repeatedly (§23.4.3.5) to reach a set of class relationships that are sufficiently general, flexible, and stable to be of real help in the further evolution of a system.

The best tool for finding initial key concepts/classes is a blackboard. The best method for their initial refinement is discussions with experts in the application domain and a couple of friends. Discussion is necessary to develop a viable initial vocabulary and conceptual framework. Few people can do that alone. One way to evolve a set of useful classes from an initial set of candidates is to simulate a system, with designers taking the roles of classes. This brings the inevitable absurdities of the initial ideas out into the open, stimulates discussion of alternatives, and creates a shared understanding of the evolving design. This activity can be supported by and documented by notes on index cards. Such cards are usually called CRC cards (''Class, Responsibility, and Collaborators''; [Wirfs-Brock,1990]) because of the information they record.

A *use case* is a description of a particular use of a system. Here is a simple example of a use case for a telephony system: take the phone off hook, dial a number, the phone at the other end rings, the phone at the other end is taken off hook. Developing a set of such use cases can be of immense value at all stages of development. Initially, finding use cases can help us understand what we are trying to build. During design, they can be used to trace a path through the system (for example, using CRC cards) to check that the relatively static description of the system in terms of classes and objects actually makes sense from a user's point of view. During programming and testing, the use cases become a source of test cases. In this way, use cases provide an orthogonal way of viewing the system and act as a reality check.

Use cases view the system as a (dynamic) working entity. They can therefore trap a designer into a functional view of a system and distract from the essential task of finding useful concepts that can be mapped into classes. Especially in the hands of someone with a background in structured analysis and weak experience with object-oriented programming/design, an emphasis on use cases can lead to a functional decomposition. A set of use cases is not a design. A focus on the use of the system must be matched by a complementary focus on the system's structure.

A team can become trapped into an inherently futile attempt to find and describe ***all*** of the use cases. This is a costly mistake. Much as when we look for candidate classes for a system, there comes a time when we must say, ''Enough is enough. The time has come to try out what we have and see what happens.'' Only by using a plausible set of classes and a plausible set of use cases in further development can we obtain the feedback that is essential to obtaining a good system. It is always hard to know when to stop a useful activity. It is especially hard to know when to stop when we know that we must return later to complete the task.

How many cases are enough? In general it is impossible to answer that question. However, in a given project, there comes a time when it is clear that most of the ordinary functioning of the system has been covered and a fair bit of the more unusual and error handling issues have been touched upon. Then it is time to get on with the next round of design and programming.

When you are trying to estimate the coverage of the system by a set of use cases, it can be useful to separate the cases into primary and secondary use cases. The primary ones describe the system's most common and ''normal'' actions, and the secondary describe the more unusual and error-handling scenarios. An example of a secondary use case would be a variant of the ''make a phone call'' case, in which the called phone is off hook, dialing its caller. It is often said that when 80% of the primary use cases and some of the secondary ones have been covered, it is time to proceed, but since we cannot know what constitutes ''all of the cases'' in advance, this is simply a rule of thumb. Experience and good sense matter here.

The concepts, operations, and relationships mentioned here are the ones that come naturally from our understanding of the application area or that arise from further work on the class structure. They represent our fundamental understanding of the application. Often, they are classifications of the fundamental concepts. For example, a hook-and-ladder is a fire engine, which is a truck, which is a vehicle. Sections §23.4.3.2 and §23.4.5 explain a few ways of looking at classes and class hierarchies with the view of making improvements.

Beware of viewgraph engineering! At some stage, you will be asked to present the design to someone and you will produce a set of diagrams explaining the structure of the system being built. This can be a very useful exercise because it helps focus your attention on what is important about the system and forces you to express your ideas in terms that others can understand. A presentation

is an invaluable design tool. Preparing a presentation with the aim of conveying real understanding to people with the interest and ability to produce constructive criticism is an exercise in conceptualization and clean expression of ideas.

However, a formal presentation of a design is also a very dangerous activity because there is a strong temptation to present an ideal system – a system you wished you could build, a system your high management wish they had – rather than what you have and what you might possibly produce in a reasonable time. When different approaches compete and executives don't really understand or care about ''the details,'' presentations can become lying competitions, in which the team that presents the most grandiose system gets to keep its job. In such cases, clear expression of ideas is often replaced by heavy jargon and acronyms. If you are a listener to such a presentation – and especially if you are a decision maker and you control development resources – it is desperately important that you distinguish wishful thinking from realistic planning. High-quality presentation materials are no guarantee of quality of the system described. In fact, I have often found that organizations that focus on the real problems get caught short when it comes to presenting their results compared to organizations that are less concerned with the production of real systems.

When looking for concepts to represent as classes, note that there are important properties of a system that cannot be represented as classes. For example, reliability, performance, and testability are important measurable properties of a system. However, even the most thoroughly object-oriented system will not have its reliability localized in a reliability object. Pervasive properties of a system can be specified, designed for, and eventually verified through measurement. Concern for such properties must be applied across all classes and may be reflected in rules for the design and implementation of individual classes and components (§23.4.3).

23.4.3.2 Step 2: Specify Operations [design.operations]

Refine the classes by specifying the sets of operations on them. Naturally, it is not possible to separate finding the classes from figuring out what operations are needed on them. However, there is a practical difference in that finding the classes focusses on the key concepts and deliberately deemphasizes the computational aspects of the classes, whereas specifying the operations focusses on finding a complete and usable set of operations. It is most often too hard to consider both at the same time, especially since related classes should be designed together. When it is time to consider both together, CRC cards (§23.4.3.1) are often helpful.

In considering what functions are to be provided, several philosophies are possible. I suggest the following strategy:

[1] Consider how an object of the class is to be constructed, copied (if at all), and destroyed.

[2] Define the *minimal* set of operations required by the concept the class is representing. Typically, these operations become the member functions (§10.3).

[3] Consider which operations could be added for notational convenience. Include only a few really important ones. Often, these operations become the nonmember ''helper functions'' (§10.3.2).

[4] Consider which operations are to be virtual, that is, operations for which the class can act as an interface for an implementation supplied by a derived class.

[5] Consider what commonality of naming and functionality can be achieved across all the classes of the component.

This is clearly a statement of minimalism. It is far easier to add every function that could conceivably be useful and to make all operations virtual. However, the more functions, the more likely they are to remain unused and the more likely they are to constrain the implementation and the further evolution of the system. In particular, functions that directly read or write part of the state of an object of a class often constrain the class to a single implementation strategy and severely limit the potential for redesign. Such functions lower the level of abstraction from a concept to one implementation of it. Adding functions also causes more work for the implementer – and for the designer in the next redesign. It is *much* easier to add a function once the need for it has been clearly established than to remove it once it has become a liability.

The reason for requiring that the decision to make a function virtual be explicit rather than a default or an implementation detail is that making a function virtual critically affects the use of its class and the relationships between that class and other classes. Objects of a class with even a single virtual function have a nontrivial layout compared to objects in languages such as C and Fortran. A class with even a single virtual function potentially acts as the interface to yet-to-be-defined classes, and a virtual function implies a dependency on yet-to-be-defined classes (§24.3.2.1).

Note that minimalism requires more work from the designer, rather than less.

When choosing operations, it is important to focus on what is to be done rather than how it is to be done. That is, we should focus more on desired behavior than on implementation issues.

It is sometimes useful to classify operations on a class in terms of their use of the internal state of objects:

- Foundation operators: constructors, destructors and copy operators
- Inspectors: operations that do not modify the state of an object
- Modifiers: operations that do modify the state of an object
- Conversions: operations that produce an object of another type based on the value (state) of the object to which they are applied
- Iterators: operations that allow access to or use of a sequence of contained objects

These categories are not orthogonal. For example, an iterator can be designed to be either an inspector or a modifier. These categories are simply a classification that has helped people approach the design of class interfaces. Naturally, other classifications are possible. Such classifications are especially useful for maintaining consistency across a set of classes within a component.

C++ provides support for the distinction between inspectors and modifiers in the form of ***const*** and non-***const*** member functions. Similarly, the notions of constructors, destructors, copy operations, and conversion functions are directly supported.

23.4.3.3 Step 3: Specify Dependencies [design.dependencies]

Refine the classes by specifying their dependencies. The various dependencies are discussed in §24.3. The key ones to consider in the context of design are parameterization, *inheritance*, and *use* relationships. Each involves consideration of what it means for a class to be responsible for a single property of a system. To be responsible certainly doesn't mean that the class has to hold all the data itself or that its member functions have to perform all the necessary operations directly. On the contrary, each class having a single area of responsibility ensures that much of the work of a class is done by directing requests ''elsewhere'' for handling by some other class that has that particular subtask as its responsibility. However, be warned that overuse of this technique can lead to

inefficient and incomprehensible designs by proliferating classes and objects to the point where no work is done except by a cascade of forwarded requests for service. What *can* be done here and now, should be.

The need to consider inheritance and use relationships at the design stage (and not just during implementation) follows directly from the use of classes to represent concepts. It also implies that the component (§23.4.3, §24.4), and not the individual class, is the unit of design.

Parameterization – often leading to the use of templates – is a way of making implicit dependencies explicit so that several alternatives can be represented without adding new concepts. Often, there is a choice between leaving something as a dependency on a context, representing it as a branch of an inheritance tree, or using a parameter (§24.4.1).

23.4.3.4 Step 4: Specify Interfaces [design.interfaces]

Specify the interfaces. Private functions don't usually need to be considered at the design stage. What implementation issues must be considered in the design stage are best dealt with as part of the consideration of dependencies in Step 2. Stronger: I use as a rule of thumb that unless at least two significantly different implementations of a class are possible, then there is probably something wrong with the class. That is, it is simply an implementation in disguise and not a representation of a proper concept. In many cases, considering if some form of lazy evaluation is feasible for a class is a good way of approaching the question, ''Is the interface to this class sufficiently implementation-independent?''

Note that public bases and friends are part of the public interface of a class; see also §11.5 and §24.4.2. Providing separate interfaces for inheriting and general clients by defining separate protected and public interfaces can be a rewarding exercise.

This is the step where the exact types of arguments are considered and specified. The ideal is to have as many interfaces as possible statically typed with application-level types; see §24.2.3 and §24.4.2.

When specifying the interfaces, look out for classes where the operations seem to support more than one level of abstraction. For example, some member functions of a class ***File*** may take arguments of type ***File_descriptor*** and others string arguments that are meant to be file names. The ***File_descriptor*** operations operate on a different level of abstraction than do the file name operations, so one must wonder whether they belong in the same class. Maybe it would be better to have two file classes, one supporting the notion of a file descriptor and another supporting the notion of a file name. Typically, all operations on a class should support the same level of abstraction. When they don't, a reorganization of the class and related classes should be considered.

23.4.3.5 Reorganization of Class Hierarchies [design.hier]

In Step 1 and again in Step 3, we examine the classes and class hierarchies to see if they adequately serve our needs. Typically they don't, and we have to reorganize to improve that structure or a design and/or an implementation.

The most common reorganizations of a class hierarchy are factoring the common part of two classes into a new class and splitting a class into two new ones. In both cases, the result is three classes: a base class and two derived classes. When should such reorganizations be done? What are common indicators that such a reorganization might be useful?

Unfortunately, there are no simple, general answers to such questions. This is not really surprising because what we are talking about are not minor implementation details, but changes to the basic concepts of a system. The fundamental – and nontrivial – operation is to look for commonality between classes and factor out the common part. The exact criteria for commonality are undefined but should reflect commonality in the concepts of the system, not just implementation conveniences. Clues that two or more classes have commonality that might be factored out into a common base class are common patterns of use, similarity of sets of operations, similarity of implementations, and simply that these classes often turn up together in design discussions. Conversely, a class might be a good candidate for splitting into two if subsets of the operations of that class have distinct usage patterns, if such subsets access separate subsets of the representation, and if the class turns up in apparently unrelated design discussions. Sometimes, making a set of related classes into a template is a way of providing necessary alternatives in a systematic manner (§24.4.1).

Because of the close relationship between classes and concepts, problems with the organization of a class hierarchy often surface as problems with the naming of classes and the use of class names in design discussions. If design discussion using class names and the classification implied by the class hierarchies sounds awkward, then there is probably an opportunity to improve the hierarchies. Note that I'm implying that two people are much better at analyzing a class hierarchy than is one. Should you happen to be without someone with whom to discuss a design, then writing a tutorial description of the design using the class names can be a useful alternative.

One of the most important aims of a design is to provide interfaces that can remain stable in the face of changes (§23.4.2). Often, this is best achieved by making a class on which many classes and functions depend into an abstract class presenting very general operations. Details are best relegated to more specialized derived classes on which fewer classes and functions directly depend. Stronger: the more classes that depend on a class, the more general that class should be and the fewer details it should reveal.

There is a strong temptation to add operations (and data) to a class used by many. This is often seen as a way of making that class more useful and less likely to need (further) change. The effect of such thinking is a class with a fat interface (§24.4.3) and with data members supporting several weakly related functions. This again implies that the class must be modified whenever there is a significant change to one of the many classes it supports. This, in turn, implies changes to apparently unrelated user classes and derived classes. Instead of complicating a class that is central to a design, we should usually keep it general and abstract. When necessary, specialized facilities should be presented as derived classes. See [Martin,1995] for examples.

This line of thought leads to hierarchies of abstract classes, with the classes near the roots being the most general and having the most other classes and functions dependent on them. The leaf classes are the most specialized and have only very few pieces of code depending directly on them. As an example, consider the final version of the ***Ival_box*** hierarchy (§12.4.3, §12.4.4).

23.4.3.6 Use of Models [design.model]

When I write an article, I try to find a suitable model to follow. That is, rather than immediately starting to type I look for papers on a similar topic to see if I can find one that can be an initial pattern for my paper. If the model I choose is a paper I wrote myself on a related topic, I might even be able to leave parts of the text in place, modify other parts as needed, and add new information

only where the logic of the information I'm trying to convey requires it. For example, this book is written that way based on its first and second editions. An extreme form of this writing technique is the form letter. In that case, I simply fill in a name and maybe add a few lines to ''personalize'' the letter. In essence, I'm writing such letters by specifying the differences from a basic model.

Such use of existing systems as models for new designs is the norm rather than the exception in all forms of creative endeavors. Whenever possible, design and programming should be based on previous work. This limits the degrees of freedom that the designer has to deal with and allows attention to be focussed on a few issues at a time. Starting a major project ''completely from scratch'' can be exhilarating. However, often a more accurate description is ''intoxicating'' and the result is a drunkard's walk through the design alternatives. Having a model is not constraining and does not require that the model should be slavishly followed; it simply frees the designer to consider one aspect of a design at a time.

Note that the use of models is inevitable because any design will be synthesized from the experiences of its designers. Having an explicit model makes the choice of a model a conscious decision, makes assumptions explicit, defines a common vocabulary, provides an initial framework for the design, and increases the likelihood that the designers have a common approach.

Naturally, the choice of an initial model is in itself an important design decision and often can be made only after a search for potential models and careful evaluation of alternatives. Furthermore, in many cases a model is suitable only with the understanding that major modification is necessary to adapt the ideas to a particular new application. Software design is hard, and we need all the help we can get. We should not reject the use of models out of misplaced disdain for ''imitation.'' Imitation is the sincerest form of flattery, and the use of models and previous work as inspiration is – within the bounds of propriety and copyright law – acceptable technique for innovative work in all fields: what was good enough for Shakespeare is good enough for us. Some people refer to such use of models in design as ''design reuse.''

Documenting general elements that turn up in many designs together with some description of the design problem they solve and the conditions under which they can be used is an obvious idea – at least once you think of it. The word *pattern* is often used to describe such a general and useful design element, and a literature exists documenting patterns and their use (for example, [Gamma,1994] and [Coplien,1995]).

It is a good idea for a designer to be acquainted with popular patterns in a given application domain. As a programmer, I prefer patterns that have some code associated with them as concrete examples. Like most people, I understand a general idea (in this case, a pattern) best when I have a concrete example (in this case, a piece of code illustrating a use of the pattern) to help me. People who use patterns heavily have a specialized vocabulary to ease communication among themselves. Unfortunately, this can become a private language that effectively excludes outsiders from understanding. As always, it is essential to ensure proper communication among people involved in different parts of a project (§23.3) and also with the design and programming communities at large.

Every successful large system is a redesign of a somewhat smaller working system. I know of no exceptions to this rule. The closest I can think of are projects that failed, muddled on for years at great cost, and then eventually became successes years after their intended completion date. Such projects unintentionally – and often unacknowledged – simply first built a nonworking system, then transformed that into a working system, and finally redesigned that into a system that approximated the original aims. This implies that it is a folly to set out to build a large system from

scratch exactly right according to the latest principles. The larger and the more ambitious a system we aim for, the more important it is to have a model from which to work. For a large system, the only really acceptable model is a somewhat smaller, related *working* system.

23.4.4 Experimentation and Analysis [design.experiment]

At the start of an ambitious development project, we do not know the best way to structure the system. Often, we don't even know precisely what the system should do because particulars will become clear only through the effort of building, testing, and using the system. How – short of building the complete system – do we get the information necessary to understand what design decisions are significant and to estimate their ramifications?

We conduct experiments. Also, we analyze the design and implementation as soon as we have something to analyze. Most frequently and importantly, we discuss the design and implementation alternatives. In all but the rarest cases, design is a social activity in which designs are developed through presentations and discussions. Often, the most important design tool is a blackboard; without it, the embryonic concepts of a design cannot be developed and shared among designers and programmers.

The most popular form of experiment seems to be to build a prototype, that is, a scaled-down version of the system or a part of the system. A prototype doesn't have stringent performance criteria, machine and programming-environment resources are typically ample, and the designers and programmers tend to be uncommonly well educated, experienced, and motivated. The idea is to get a version running as fast as possible to enable exploration of design and implementation choices.

This approach can be very successful when done well. It can also be an excuse for sloppiness. The problem is that the emphasis of a prototype can easily shift from ''exploring design alternatives'' to ''getting some sort of system running as soon as possible.'' This easily leads to a disinterest in the internal structure of the prototype (''after all, it is only a prototype'') and a neglect of the design effort in favor of playing around with the prototype implementation. The snag is that such an implementation can degenerate into the worst kind of resource hog and maintenance nightmare while giving the illusion of an ''almost complete'' system. Almost by definition, a prototype does not have the internal structure, the efficiency, and the maintenance infrastructure that allows it to scale to real use. Consequently, a ''prototype'' that becomes an ''almost product'' soaks up time and energy that could have been better spent on the product. The temptation for both developers and managers is to make the prototype into a product and postpone ''performance engineering'' until the next release. Misused this way, prototyping is the negation of all that design stands for.

A related problem is that the prototype developers can fall in love with their tools. They can forget that the expense of their (necessary) convenience cannot always be afforded by a production system and that the freedom from constraints and formalities offered by their small research group cannot easily be maintained for a larger group working toward a set of interlocking deadlines.

On the other hand, prototypes can be invaluable. Consider designing a user interface. In this case, the internal structure of the part of the system that doesn't interact directly with the user often *is* irrelevant and there are no other feasible ways of getting experience with users' reactions to the look and feel of a system. Another example is a prototype designed strictly for studying the internal workings of a system. Here, the user interface can be rudimentary – possibly with simulated users instead of real ones.

Prototyping is a way of experimenting. The desired results from building a prototype are the insights that building it brings, not the prototype itself. Maybe the most important criterion for a prototype is that it has to be so incomplete that it is obviously an experimental vehicle and cannot be turned into a product without a major redesign and reimplementation. Having a prototype ''incomplete'' helps keep the focus on the experiment and minimizes the danger of having the prototype become a product. It also minimizes the temptation to try to base the design of the product too closely on the design of the prototype – thus forgetting or ignoring the inherent limitations of the prototype. After use, a prototype should be thrown away.

It should be remembered that in many cases, there are experimental techniques that can be used as alternatives to prototyping. Where those can be used, they are often preferable because of their greater rigor and lower demands on designer time and system resources. Examples are mathematical models and various forms of simulators. In fact, one can see a continuum from mathematical models, through more and more detailed simulations, through prototypes, through partial implementations, to a complete system.

This leads to the idea of growing a system from an initial design and implementation through repeated redesign and reimplementation. This is the ideal strategy, but it can be very demanding on design and implementation tools. Also, the approach suffers from the risk of getting burdened with so much code reflecting initial design decisions that a better design cannot be implemented. At least for now, this strategy seems limited to small-to-medium-scale projects, in which major changes to the overall design are unlikely, and for redesigns and reimplementations after the initial release of the system, where such a strategy is inevitable.

In addition to experiments designed to provide insights into design choices, analysis of a design and/or an implementation itself can be an important source of further insights. For example, studies of the various dependencies between classes (§24.3) can be most helpful, and traditional implementer's tools such as call graphs, performance measurements, etc., must not be ignored.

Note that specifications (the output of the analysis phase) and designs are as prone to errors as is the implementation. In fact, they may be more so because they are even less concrete, are often specified less precisely, are not executable, and typically are not supported by tools of a sophistication comparable to what is available for checking and analyzing the implementation. Increasing the formality of the language/notation used to express a design can go some way toward enabling the application of tools to help the designer. This must not be done at the cost of impoverishing the programming language used for implementation (§24.3.1). Also, a formal notation can itself be a source of complexity and problems. This happens when the formalism is ill suited to the practical problem to which it is applied, when the rigor of the formalism exceeds the mathematical background and maturity of the designers and programmers involved, and when the formal description of a system gets out of touch with the system it is supposedly describing.

Design is inherently error-prone and hard to support with effective tools. This makes experience and feedback essential. Consequently, it is fundamentally flawed to consider the software-development process a linear process starting with analysis and ending with testing. An emphasis on iterative design and implementation is needed to gain sufficient feedback from experience during the various stages of development.

23.4.5 Testing [design.test]

A program that has not been tested does not work. The ideal of designing and/or verifying a program so that it works the first time is unattainable for all but the most trivial programs. We should strive toward that ideal, but we should not be fooled into thinking that testing is easy.

''How to test?'' is a question that cannot be answered in general. ''When to test?'' however, does have a general answer: as early and as often as possible. Test strategies should be generated as part of the design and implementation efforts or at least should be developed in parallel with them. As soon as there is a running system, testing should begin. Postponing serious testing until ''after the implementation is complete'' is a prescription for slipped schedules and/or flawed releases.

Wherever possible, a system should be designed specifically so that it is relatively easy to test. In particular, mechanisms for testing can often be designed right into the system. Sometimes this is not done out of fear of causing expensive run-time testing or for fear that the redundancy necessary for consistency checks will unduly enlarge data structures. Such fear is usually misplaced because most actual testing code and redundancy can, if necessary, be stripped out of the code before the system is shipped. Assertions (§24.3.7.2) are sometimes useful here.

More important than specific tests is the idea that the structure of the system should be such that we have a reasonable chance of convincing ourselves and our users/customers that we can eliminate errors by a combination of static checking, static analysis, and testing. Where a strategy for fault tolerance is developed (§14.9), a testing strategy can usually be designed as a complementary and closely related aspect of the total design.

If testing issues are completely discounted in the design phase, then testing, delivery date, and maintenance problems will result. The class interfaces and the class dependencies (as described in §24.3 and §24.4.2) are usually a good place to start work on a testing strategy.

Determining how much testing is enough is usually hard. However, too little testing is a more common problem than too much. Exactly how many resources should be allocated to testing compared to design and implementation naturally depends on the nature of the system and the methods used to construct it. However, as a rule of thumb, I can suggest that more resources in time, effort, and talent should be spent testing a system than on constructing the initial implementation. Testing should focus on problems that would have disastrous consequences and on problems that would occur frequently.

23.4.6 Software Maintenance [design.maintain]

''Software maintenance'' is a misnomer. The word ''maintenance'' suggests a misleading analogy to hardware. Software doesn't need oiling, doesn't have moving parts that wear down, and doesn't have crevices in which water can collect and cause rust. Software can be replicated *exactly* and transported over long distances at minute costs. Software is not hardware.

The activities that go under the name of software maintenance are really redesign and reimplementation and thus belong under the usual program development cycle. When flexibility, extensibility, and portability are emphasized in the design, the traditional sources of maintenance problems are addressed directly.

Like testing, maintenance must not be an afterthought or an activity segregated from the mainstream of development. In particular, it is important to have some continuity in the group of people

involved in a project. It is not easy to successfully transfer maintenance to a new (and typically less-experienced) group of people with no links to the original designers and implementers. When a major change of people is necessary, there must be an emphasis on transferring an understanding of the system's structure and of the system's aims to the new people. If a ''maintenance crew'' is left guessing about the architecture of the system or must deduce the purpose of system components from their implementation, the structure of a system can deteriorate rapidly under the impact of local patches. Documentation is typically much better at conveying details than in helping new people to understand key ideas and principles.

23.4.7 Efficiency [design.efficiency]

Donald Knuth observed that ''premature optimization is the root of all evil.'' Some people have learned that lesson all too well and consider all concern for efficiency evil. On the contrary, efficiency must be kept in mind throughout the design and implementation effort. However, that does not mean the designer should be concerned with micro-efficiencies, but that first-order efficiency issues must be considered.

The best strategy for efficiency is to produce a clean and simple design. Only such a design can remain relatively stable over the lifetime of the project and serve as a base for performance tuning. Avoiding the gargantuanism that plagues large projects is essential. Far too often people add features ''just in case'' (§23.4.3.2, §23.5.3) and end up doubling and quadrupling the size and runtime of systems to support frills. Worse, such overelaborate systems are often unnecessarily hard to analyze so that it becomes difficult to distinguish the avoidable overheads from the unavoidable. Thus, even basic analysis and optimization is discouraged. Optimization should be the result of analysis and performance measurement, not random fiddling with the code. Especially in larger systems, a designer's or programmer's ''intuition'' is an unreliable guide in matters of efficiency.

It is important to avoid inherently inefficient constructs and constructs that will take much time and cleverness to optimize to an acceptable performance level. Similarly, it is important to minimize the use of inherently nonportable constructs and tools because using such tools and constructs condemns the project to run on older (less powerful and/or more expensive) computers.

23.5 Management [design.management]

Provided it makes some minimum of sense, most people do what they are encouraged to do. In particular, if in the context of a software project you reward certain ways of operating and penalize others, only exceptional programmers and designers will risk their careers to do what they consider right in the face of management opposition, indifference, and red tape†. It follows that an organization should have a reward structure that matches its stated aims of design and programming. However, all too often this is not the case: a major change of programming style can be achieved only through a matching change of design style, and both typically require changes in management style to be effective. Mental and organizational inertia all too easily leads to a local change that is not

† An organization that treats its programmers as morons will soon have programmers that are willing and able to act like morons only.

supported by global changes required to ensure its success. A fairly typical example is a change to a language that supports object-oriented programming, such as C++, without a matching change in the design strategies to take advantage of its facilities (see also §24.2). Another is a change to ''object-oriented design'' without the introduction of a programming language to support it.

23.5.1 Reuse [design.reuse]

Increased reuse of code and design is often cited as a major reason for adopting a new programming language or design strategy. However, most organizations reward individuals and groups that choose to re-invent the wheel. For example, a programmer may have his productivity measured in lines of code; will he produce small programs relying on standard libraries at the cost of income and, possibly, status? A manager may be paid somewhat proportionally to the number of people in her group; is she going to use software produced in another group when she can hire another couple of programmers for her own group instead? A company can be awarded a government contract, where the profit is a fixed percentage of the development cost; is that company going to minimize its profits by using the most effective development tools? Rewarding reuse is hard, but unless management finds ways to encourage and reward it, reuse will not happen.

Reuse is primarily a social phenomenon. I can use someone else's software provided that:

[1] It works: to be reusable, software must first be usable.
[2] It is comprehensible: program structure, comments, documentation, and tutorial material are important.
[3] It can coexist with software not specifically written to coexist with it.
[4] It is supported (or I'm willing to support it myself; typically, I'm not).
[5] It is economical (can I share the development and maintenance costs with other users?).
[6] I can find it.

To this, we may add that a component is not reusable until someone has ''reused'' it. The task of fitting a component into an environment typically leads to refinements in its operation, generalizations of its behavior, and improvements in its ability to coexist with other software. Until this exercise has been done at least once, even components that have been designed and implemented with the greatest care tend to have unintended and unexpected rough corners.

My experience is that the conditions necessary for reuse will exist only if someone makes it their business to make such sharing work. In a small group, this typically means that an individual, by design or by accident, becomes the keeper of common libraries and documentation. In a larger organization, this means that a group or department is chartered to gather, build, document, popularize, and maintain software for use by many groups.

The importance of such a ''standard components'' group cannot be overestimated. Note that as a first approximation, a system reflects the organization that produced it. If an organization has no mechanism for promoting and rewarding cooperation and sharing, cooperation and sharing will be rare. A standard components group must actively promote its components. This implies that good traditional documentation is essential but insufficient. In addition, the components group must provide tutorials and other information that allow a potential user to find a component and understand why it might be of help. This implies that activities that traditionally are associated with marketing and education must be undertaken by the components group.

Whenever possible, the members of this group should work in close cooperation with applications builders. Only then can they be sufficiently aware of the needs of users and alert to the opportunities for sharing components among different applications. This argues for there to be a consultancy role for such an organization and for the use of internships to transfer information into and out of the components group.

The success of a ''components group'' must be measured in terms of the success of its clients. If its success is measured simply in terms of the amount of tools and services it can convince development organizations to accept, such a group can become corrupted into a mere peddler of commercial software and a proponent of ever-changing fads.

Not all code needs to be reusable, and reusability is not a universal property. Saying that a component is ''reusable'' means that its reuse within a certain framework requires little or no work. In most cases, moving to a different framework will require significant work. In this respect, reuse strongly resembles portability. It is important to note that reuse is the result of design aimed at reuse, refinement of components based on experience, and deliberate effort to search out existing components to (re)use. Reuse does not magically arise from mindless use of specific language features or coding techniques. C++ features such as classes, virtual functions, and templates allow designs to be expressed so that reuse is made easier (and thus more likely), but in themselves such features do not ensure reusability.

23.5.2 Scale [design.scale]

It is easy for an individual or an organization to get excited about ''doing things right.'' In an institutional setting, this often translates into ''developing and strictly following proper procedures.'' In both cases, common sense can be the first victim of a genuine and often ardent desire to improve the way things are done. Unfortunately, once common sense is missing there is no limit to the damage that can unwittingly be done.

Consider the stages of the development process listed in §23.4 and the stages of the design steps listed in §23.4.3. It is relatively easy to elaborate these stages into a proper design method where each stage is more precisely defined and has well-defined inputs and outputs and a semiformal notation for expressing these inputs and outputs. Checklists can be developed to ensure that the design method is adhered to, and tools can be developed to enforce a large number of the procedural and notational conventions. Further, looking at the classification of dependencies presented in §24.3 one could decree that certain dependencies were good and others bad and provide analysis tools to ensure that these value judgements were applied uniformly across a project. To complete this ''firming up'' of the software-production process, one would define standards for documentation (including rules for spelling and grammar and typesetting conventions) and for the general look of the code (including specifications of which language features can and cannot be used, specifications of what kinds of libraries can and cannot be used, conventions for indentation and the naming of functions, variables, and types, etc.).

Much of this can be helpful for the success of a project. At least, it would be a folly to set out to design a system that will eventually contain ten million lines of code that will be developed by hundreds of people and maintained and supported by thousands more over a decade or more without a fairly well-defined and somewhat rigid framework along the lines described previously.

Fortunately, most systems do not fall into this category. However, once the idea is accepted that such a design method or adherence to such a set of coding and documentation standards is ''the right way,'' pressure builds to apply it universally and in every detail. This can lead to ludicrous constraints and overheads on small projects. In particular, it can lead to paper shuffling and forms filling replacing productive work as the measure of progress and success. If that happens, real designers and programmers will leave the project and be replaced with bureaucrats.

Once such a ridiculous misapplication of a (hopefully perfectly reasonable) design method has occurred in a community, its failure becomes the excuse for avoiding almost all formality in the development process. This in turn naturally leads to the kind of messes and failures that the design method was designed to prevent in the first place.

The real problem is to find an appropriate degree of formality for the development of a particular project. Don't expect to find an easy answer to this problem. Essentially every approach works for a small project. Worse, it seems that essentially every approach – however ill conceived and however cruel to the individuals involved – also works for a large project, provided you are willing to throw indecent amounts of time and money at the problem.

A key problem in every software project is how to maintain the integrity of the design. This problem increases more than linearly with scale. Only an individual or a small group of people can grasp and keep sight of the overall aims of a major project. Most people must spend so much of their time on subprojects, technical details, day-to-day administration, etc., that the overall design aims are easily forgotten or subordinated to more local and immediate goals. It also is a recipe for failure not to have an individual or group with the explicit task of maintaining the integrity of the design. It is a recipe for failure not to enable such an individual or group to have an effect on the project as a whole.

Lack of a consistent long-term aim is much more damaging to a project and an organization than the lack of any individual feature. It should be the job of some small number of individuals to formulate such an overall aim, to keep that aim in mind, to write the key overall design documents, to write the introductions to the key concepts, and generally to help others to keep the overall aim in mind.

23.5.3 Individuals [design.people]

Use of design as described here places a premium on skillful designers and programmers. Thus, it makes the choice of designers and programmers critical to the success of an organization.

Managers often forget that organizations consist of individuals. A popular notion is that programmers are equal and interchangeable. This is a fallacy that can destroy an organization by driving out many of the most effective individuals and condemning the remaining people to work at levels well below their potential. Individuals are interchangeable only if they are not allowed to take advantage of skills that raise them above the absolute minimum required for the task in question. Thus, the fiction of interchangeability is inhumane and inherently wasteful.

Most programming performance measures encourage wasteful practices and fail to take critical individual contributions into account. The most obvious example is the relatively widespread practice of measuring progress in terms of number of lines of code produced, number of pages of documentation produced, number of tests passed, etc. Such figures look good on management charts but bear only the most tenuous relation to reality. For example, if productivity is measured in terms

of number of lines of code produced, a successful application of reuse will appear to cause negative performance of programmers. A successful application of the best principles in the redesign of a major piece of software typically has the same effect.

Quality of work produced is far harder to measure than quantity of output, yet individuals and groups must be rewarded based on the quality of their output rather than by crude quantity measures. Unfortunately, the design of practical quality measures has – to the best of my knowledge – hardly begun. In addition, measures that incompletely describe the state of a project tend to warp development. People adapt to meet local deadlines and to optimize individual and group performance as defined by the measures. As a direct result, overall system integrity and performance suffer. For example, if a deadline is defined in terms of bugs removed or known bugs remaining, we may see that deadline met at the expense of run-time performance or hardware resources needed to run the system. Conversely, if only run-time performance is measured the error rate will surely rise when the developers struggle to optimize the system for benchmarks. The lack of good and comprehensive quality measures places great demands on the technical expertise of managers, but the alternative is a systematic tendency to reward random activity rather than progress. Don't forget that managers are also individuals. Managers need as least as much education on new techniques as do the people they manage.

As in other areas of software development, we must consider the longer term. It is essentially impossible to judge the performance of an individual on the basis of a single year's work. Most individuals do, however, have consistent long-term track records that can be reliable predictors of technical judgement and a useful help in evaluating immediate past performance. Disregard of such records – as is done when individuals are considered merely as interchangeable cogs in the wheels of an organization – leaves managers at the mercy of misleading quantity measurements.

One consequence of taking a long-term view and avoiding the ''interchangeable morons school of management'' is that individuals (both developers and managers) need longer to grow into the more demanding and interesting jobs. This discourages job hopping as well as job rotation for ''career development.'' A low turnover of both key technical people and key managers must be a goal. No manager can succeed without a rapport with key designers and programmers and some recent and relevant technical knowledge. Conversely, no group of designers and developers can succeed in the long run without support from competent managers and a minimum of understanding of the larger nontechnical context in which they work.

Where innovation is needed, senior technical people, analysts, designers, programmers, etc., have a critical and difficult role to play in the introduction of new techniques. These are the people who must learn new techniques and in many cases unlearn old habits. This is not easy. These individuals have typically made great personal investments in the old ways of doing things and rely on successes achieved using these ways of operating for their technical reputation. So do many technical managers.

Naturally, there is often a fear of change among such individuals. This can lead to an overestimation of the problems involved in a change and a reluctance to acknowledge problems with the old ways of doing things. Equally naturally, people arguing for change tend to overestimate the beneficial effects of new ways of doing things and to underestimate the problems involved in a change. These two groups of individuals *must* communicate, they *must* learn to talk the same language, they *must* help each other hammer out a model for transition. The alternative is organizational paralysis and the departure of the most capable individuals from both groups. Both groups

should remember that the most successful ''old timers'' are often the ''young turks'' of yesteryear. Given a chance to learn without humiliation, more experienced programmers and designers can become the most successful and insightful proponents of change. Their healthy skepticism, knowledge of users, and acquaintance with the organizational hurdles can be invaluable. Proponents of immediate and radical change must realize that a transition, often involving a gradual adoption of new techniques, is more often than not necessary. Conversely, individuals who have no desire to change should search out areas in which no change is needed rather than fight vicious rear-guard battles in areas in which new demands have already significantly altered the conditions for success.

23.5.4 Hybrid Design [design.hybrid]

Introducing new ways of doing things into an organization can be painful. The disruption to the organization and the individuals in the organization can be significant. In particular, an abrupt change that overnight turns productive and proficient members of ''the old school'' into ineffective novices in ''the new school'' is typically unacceptable. However, it is rare to achieve major gains without changes, and significant changes typically involve risks.

C++ was designed to minimize such risks by allowing a gradual adoption of techniques. Although it is clear that the largest benefits from using C++ are achieved through data abstraction, object-oriented programming, and object-oriented design, it is not clear that the fastest way to achieve these gains is a radical break with the past. Occasionally, such a clean break is feasible. More often, the desire for improvement is – or should be – tempered by concerns about how to manage the transition. Consider:

- Designers and programmers need time to acquire new skills.
- New code needs to cooperate with old code.
- Old code needs to be maintained (often indefinitely).
- Work on existing designs and programs needs to be completed (on time).
- Tools supporting the new techniques need to be introduced into the local environment.

These factors lead naturally to a hybrid style of design – even where that isn't the intention of some designers. It is easy to underestimate the first two points.

By supporting several programming paradigms, C++ supports the notion of a gradual introduction into an organization in several ways:

- Programmers can remain productive while learning C++.
- C++ can yield significant benefits in a tool-poor environment.
- C++ program fragments can cooperate well with code written in C and other traditional languages.
- C++ has a large C-compatible subset.

The idea is that programmers can make the move to C++ from a traditional language by first adopting C++ while retaining a traditional (procedural) style of programming. Then they use the data abstraction techniques. Finally – when the language and its associated tools have been mastered – they move on to object-oriented programming and generic programming. Note that a well-designed library is much easier to use than it was to design and implement, so a novice can benefit from the more advanced uses of abstraction even during the early stages of this progress.

The idea of learning object-oriented design, object-oriented programming, and C++ in stages is supported by facilities for mixing C++ code with code written in languages that do not support

C++'s notions of data abstraction and object-oriented programming (§24.2.1). Many interfaces can simply be left procedural because there will be no immediate benefits in doing anything more complicated. For many key libraries, this will already have been done by the library provider so that the C++ programmer can stay ignorant of the actual implementation language. Using libraries written in languages such as C is the first, and initially most important, form of reuse in C++.

The next stage – to be used only where a more elaborate technique is actually needed – is to present facilities written in languages such as C and Fortran as classes by encapsulating the data structures and functions in C++ interface classes. A simple example of lifting the semantics from the procedure plus data structure level to the data abstraction level is the string class from §11.12. There, encapsulation of the C character string representation and the standard C string functions is used to produce a string type that is much simpler to use.

A similar technique can be used to fit a built-in or stand-alone type into a class hierarchy (§23.5.1). This allows designs for C++ to evolve to use data abstraction and class hierarchies in the presence of code written in languages in which these concepts are missing and even under the constraint that the resulting code must be callable from procedural languages.

23.6 Annotated Bibliography [design.ref]

This chapter only scratches the surface of the issues of design and of the management of programming projects. For that reason, a short annotated bibliography is provided. An extensive annotated bibliography can be found in [Booch,1994].

[Anderson,1990] Bruce Anderson and Sanjiv Gossain: *An Iterative Design Model for Reusable Object-Oriented Software*. Proc. OOPSLA'90. Ottawa, Canada. A description of an iterative design and redesign model with a specific example and a discussion of experience.

[Booch,1994] Grady Booch: *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings. 1994. ISBN 0-8053-5340-2. Contains a detailed description of design, a specific design method with a graphical notation, and several large examples of designs expressed in C++. It is an excellent book to which this chapter owes much. It provides a more in-depth treatment of many of the issues in this chapter.

[Booch,1996] Grady Booch: *Object Solutions*. Benjamin/Cummings. 1996. ISBN 0-8053-0594-7. Describes the development of object-oriented systems from a management perspective. Contains extensive C++ code examples.

[Brooks,1982] Fred Brooks: *The Mythical Man Month*. Addison-Wesley. 1982. Everyone should read this book every couple of years. A warning against hubris. It is a bit dated on technical matters, but it is not at all dated in matters related to individuals, organizations, and scale. Republished with additions in 1997. ISBN 1-201-83595-9.

[Brooks,1987] Fred Brooks: *No Silver Bullet*. IEEE Computer, Vol. 20, No. 4. April 1987. A summary of approaches to large-scale software development, with a much-needed warning against belief in miracle cures (''silver bullets'').

[Coplien,1995] James O. Coplien and Douglas C. Schmidt (editors): *Pattern Languages of Program Design*. Addison-Wesley. 1995. ISBN 1-201-60734-4.

[Gamma,1994] Eric Gamma, et. al.: *Design Patterns*. Addison-Wesley. 1994. ISBN 0-201-63361-2. A practical catalog of techniques for creating flexible and reusable software, with a nontrivial, well-explained example. Contains extensive C++ code examples.

[DeMarco,1987] T. DeMarco and T. Lister: *Peopleware*. Dorset House Publishing Co. 1987. One of the few books that focusses on the role of people in the production of software. A must for every manager. Smooth enough for bedside reading. An antidote for much silliness.

[Jacobson,1992] Ivar Jacobson et. al.: *Object-Oriented Software Engineering*. Addison-Wesley. 1992. ISBN 0-201-54435-0. A thorough and practical description of software development in an industrial setting with an emphasis on use cases (§23.4.3.1). Miscasts C++ by describing it as it was ten years ago.

[Kerr,1987] Ron Kerr: *A Materialistic View of the Software ''Engineering'' Analogy*. In SIGPLAN Notices, March 1987. The use of analogy in this chapter and the next owes much to the observations in this paper and to the presentations by and discussions with Ron that preceded it.

[Liskov,1987] Barbara Liskov: *Data Abstraction and Hierarchy*. Proc. OOPSLA'87 (Addendum). Orlando, Florida. A discussion of how the use of inheritance can compromise data abstraction. Note, C++ has specific language support to help avoid most of the problems mentioned (§24.3.4).

[Martin,1995] Robert C. Martin: *Designing Object-Oriented C++ Applications Using the Booch Method*. Prentice-Hall. 1995. ISBN 0-13-203837-4. Shows how to go from a problem to C++ code in a fairly systematic way. Presents alternative designs and principles for choosing between them. More practical and more concrete than most books on design. Contains extensive C++ code examples.

[Parkinson,1957] C. N. Parkinson: *Parkinson's Law and other Studies in Administration*. Houghton Mifflin. Boston. 1957. One of the funniest and most cutting descriptions of disasters caused by administrative processes.

[Meyer,1988] Bertrand Meyer: *Object Oriented Software Construction*. Prentice Hall. 1988. Pages 1-64 and 323-334 give a good introduction to one view of object-oriented programming and design with many sound pieces of practical advice. The rest of the book describes the Eiffel language. Tends to confuse Eiffel with universal principles.

[Shlaer,1988] S. Shlaer and S. J. Mellor: *Object-Oriented Systems Analysis* and *Object Lifecycles*. Yourdon Press. ISBN 0-13-629023-X and 0-13-629940-7. Presents a view of analysis, design, and programming that differs strongly from the one presented here and embodied in C++ and does so using a vocabulary that makes it sound rather similar.

[Snyder,1986] Alan Snyder: *Encapsulation and Inheritance in Object-Oriented Programming Languages*. Proc. OOPSLA'86. Portland, Oregon. Probably the

first good description of the interaction between encapsulation and inheritance. Also provides a nice discussion of some notions of multiple inheritance.

[Wirfs-Brock,1990] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener: *Designing Object-Oriented Software*. Prentice Hall. 1990. Describes an anthropomorphic design method based on role playing using CRC (Classes, Responsibilities, and Collaboration) cards. The text, if not the method itself, is biased toward Smalltalk.

23.7 Advice [design.advice]

[1] Know what you are trying to achieve; §23.3.
[2] Keep in mind that software development is a human activity; §23.2, §23.5.3.
[3] Proof by analogy is fraud; §23.2.
[4] Have specific and tangible aims; §23.4.
[5] Don't try technological fixes for sociological problems; §23.4.
[6] Consider the longer term in design and in the treatment of people; §23.4.1, §23.5.3.
[7] There is no lower limit to the size of programs for which it is sensible to design before starting to code; §23.2.
[8] Design processes to encourage feedback; §23.4.
[9] Don't confuse activity for progress; §23.3, §23.4.
[10] Don't generalize beyond what is needed, what you have direct experience with, and what can be tested; §23.4.1, §23.4.2.
[11] Represent concepts as classes; §23.4.2, §23.4.3.1.
[12] There are properties of a system that should not be represented as a class; §23.4.3.1.
[13] Represent hierarchical relationships between concepts as class hierarchies; §23.4.3.1.
[14] Actively search for commonality in the concepts of the application and implementation and represent the resulting more general concepts as base classes; §23.4.3.1, §23.4.3.5.
[15] Classifications in other domains are not necessarily useful classifications in an inheritance model for an application; §23.4.3.1.
[16] Design class hierarchies based on behavior and invariants; §23.4.3.1, §23.4.3.5, §24.3.7.1.
[17] Consider use cases; §23.4.3.1.
[18] Consider using CRC cards; §23.4.3.1.
[19] Use existing systems as models, as inspiration, and as starting points; §23.4.3.6.
[20] Beware of viewgraph engineering; §23.4.3.1.
[21] Throw a prototype away before it becomes a burden; §23.4.4
[22] Design for change, focusing on flexibility, extensibility, portability, and reuse; §23.4.2.
[23] Focus on component design; §23.4.3.
[24] Let each interface represent a concept at a single level of abstraction; §23.4.3.1.
[25] Design for stability in the face of change; §23.4.2.
[26] Make designs stable by making heavily-used interfaces minimal, general, and abstract; §23.4.3.2, §23.4.3.5.
[27] Keep it small. Don't add features ''just in case;'' §23.4.3.2.

[28] Always consider alternative representations for a class. If no alternative representation is plausible, the class is probably not representing a clean concept; §23.4.3.4.
[29] Repeatedly review and refine both the design and the implementation; §23.4, §23.4.3.
[30] Use the best tools available for testing and for analyzing the problem, the design, and the implementation; §23.3, §23.4.1, §23.4.4.
[31] Experiment, analyze, and test as early as possible and as often as possible; §23.4.4, §23.4.5.
[32] Don't forget about efficiency; §23.4.7.
[33] Keep the level of formality appropriate to the scale of the project; §23.5.2.
[34] Make sure that someone is in charge of the overall design; §23.5.2.
[35] Document, market, and support reusable components; §23.5.1.
[36] Document aims and principles as well as details; §23.4.6.
[37] Provide tutorials for new developers as part of the documentation; §23.4.6.
[38] Reward and encourage reuse of designs, libraries, and classes; §23.5.1.