0

Introduction

Developing good C++ programs is not easy. Developing highly reliable and maintainable software in C++ becomes even more difficult and introduces many new concepts as projects become larger. Just as experience gained from building single-family homes does not qualify a carpenter to erect a skyscraper, many techniques and practices learned through experiences with smaller C++ projects simply do not scale well to larger development efforts.

This book is about how to design very large, high-quality software systems. It is intended for experienced C++ software developers who strive to create highly maintainable, highly testable software architectures. This book is not a theoretical approach to programming; it is a thorough, practical guide to success, drawing from years of experience of expert C++ programmers developing huge, multi-site systems. We will demonstrate how to design systems that involve hundreds of programmers, thousands of classes, and potentially millions of lines of C++ source code.

This introduction considers some of the kinds of problems encountered when developing large projects in C++, and provides a context for the groundwork we must do in the early chapters. In this introduction several terms are used without definition. Most of these terms should be understandable from context. In the chapters that follow, these terms are defined more precisely. The real payoff will come in Chapter 5, where we begin to apply specific techniques to reduce the coupling (i.e., the degree of interdependency) within our C++ systems.

0.1 From C to C++

The potential advantages of the object-oriented paradigm in managing the complexity of large systems are widely assumed. As of the writing of this book, the number of C++ programmers has been doubling every seven to nine months.[1] In the hands of experienced C++ programmers, C++ is a powerful amplifier of human skill and engineering talent. It is completely wrong, however, to think that just using C++ will ensure success in a large project.

C++ is not just an extension of C: it supports an entirely new paradigm. The object-oriented paradigm is notorious for demanding more design effort and savvy than its procedural counterpart. C++ is more difficult to master than C, and there are innumerable ways to shoot yourself in the foot. Often you won't realize a serious error until it is much too late to fix it and still meet your schedule. Even relatively small indiscretions, such as the indiscriminate use of virtual functions or the passing of user-defined types by value, can result in perfectly correct C++ programs that run ten times slower than they would have had you written them in C.

During the initial exposure to C++, there is invariably a period during which productivity will grind to a halt as the seemingly limitless design alternatives are explored. During this period, conventional procedural programmers will be filled with an uneasiness as they try to get their arms around the concept referred to as *object oriented.*

Although the size and complexity of the C++ language can at first be somewhat overwhelming for even the most experienced professional C programmers, it does not take too long for a competent C programmer to get a small, nontrivial C++ program up and running. Unfortunately, the undisciplined techniques used to create small programs in C++ are totally inadequate for tackling larger projects. That is to say, a naive application of C++ technology does not scale well to larger projects. The consequences for the uninitiated are many.

0.2 Using C++ to Develop Large Projects

Just like a program in C, a poorly written C++ program can be very hard to understand and maintain. If interfaces are not fully encapsulating, it will be difficult to tune

---

[1] **stroustrup94**, Section 7.1, pp. 163–164.

or to enhance implementations. Poor encapsulation will hinder reuse, and any advantage in testability will be eliminated.

Contrary to popular belief, object-oriented programs *in their most general form* are fundamentally more difficult to test and verify than their procedural counterparts.[2] The ability to alter internal behavior via virtual functions can invalidate class invariants essential to correct performance. Further, the potential number of control flow paths through an object-oriented system can be explosively large.

Fortunately, it is not necessary to write such arbitrarily general (and untestable) object-oriented programs. Reliability can be achieved by restricting our use of the paradigm to a more testable subset.

As programs get larger, forces of a different nature come into play. The following subsections illustrate specific instances of some of the kinds of problems that we are likely to encounter.

0.2.1 Cyclic Dependencies

As a software professional, you have probably been in a situation where you were looking at a software system for the first time and you could not seem to find a reasonable starting point or a piece of the system that made sense on its own. Not being able to understand or use any part of a system independently is a symptom of a cyclically dependent design. C++ objects have a phenomenal tendency to get tangled up in each other. This insidious form of tight physical coupling is illustrated in Figure 0-1. A circuit is a collection of elements and wires. Consequently, class `Circuit` knows about the definitions of both `Element` and `Wire`. An element knows the circuit to which it belongs, and can tell whether or not it is connected to a specified wire. Hence class `Element` also knows about both `Circuit` and `Wire`. Finally, a wire can be connected to a terminal of either an element or a circuit. In order to do its job, class `Wire` must access the definitions of both `Element` and `Circuit`.

The definitions for each of these three object types reside in separate physical components (translation units) in order to improve modularity. Even though the implementations of these individual types are fully encapsulated by their interfaces, however, the `.c` files for each component are forced to include the header files of the other two.

[2] **perry**, pp. 13-19.

The resulting dependency graph for these three components is cyclic. That is, no one component can be used or even tested without the other two.

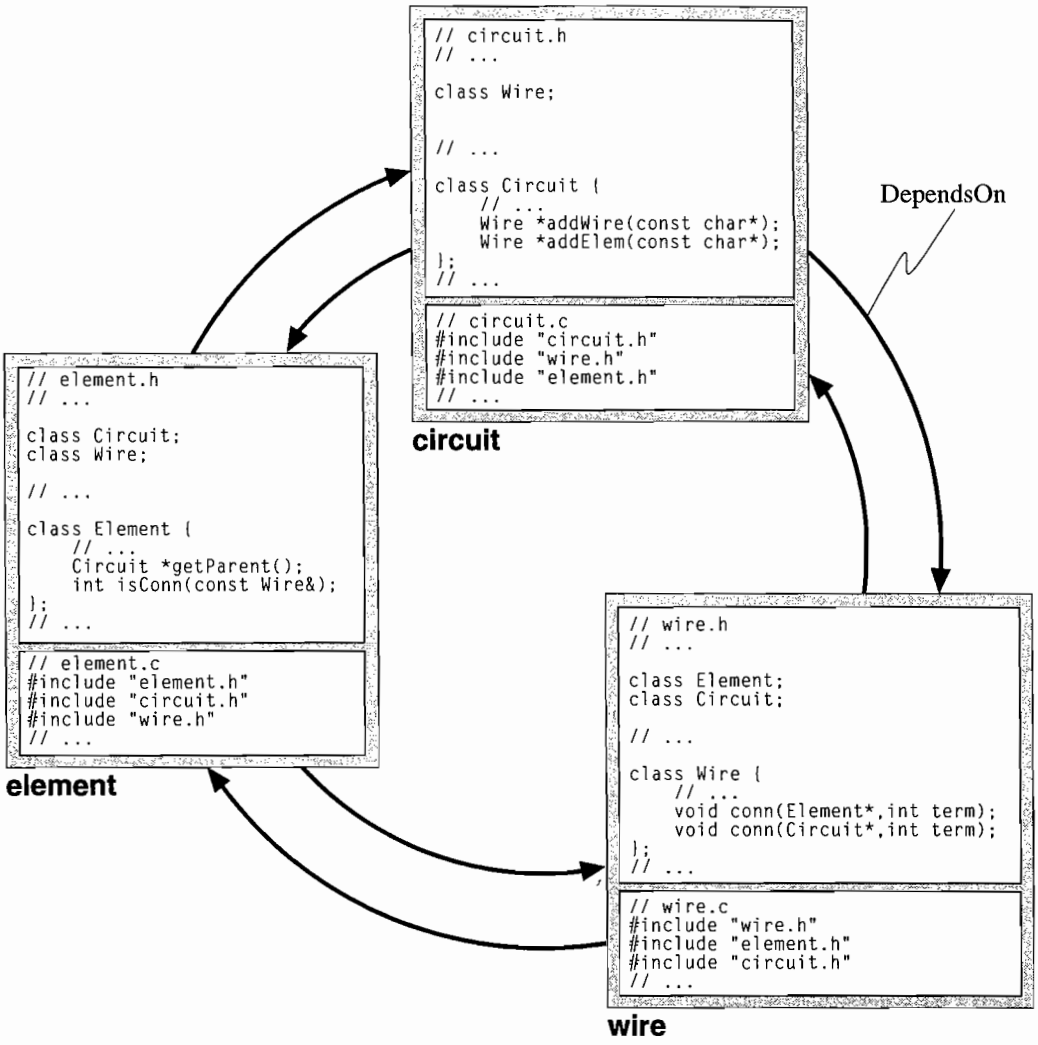


Figure 0-1: Cyclically Dependent Components

Large systems that are naively architected tend to become tightly coupled by cyclic dependencies and fiercely resist decomposition. Supporting such systems can be a nightmare, and effective modular testing is often impossible.

A case in point is an early version of an electronic-design database. At the time, its authors did not realize the need for avoiding cyclic dependencies in the physical design. The result was an interdependent collection of files containing hundreds of classes with thousands of functions, and no way to use or even test it except as a single module. This system had very poor reliability, proved impractical to extend or maintain, and ultimately had to be thrown out and rewritten from scratch.

By contrast, hierarchical physical designs (i.e., without cyclic interdependencies) are relatively easy to understand, test, and reuse incrementally.

0.2.2 Excessive Link-Time Dependencies

If you have attempted to link to a small amount of functionality in a library and found that your time to link has increased disproportionately to the benefit you are deriving, then you may have been trying to reuse heavy-weight rather than light-weight components.

One of the nice things about objects is that it is easy to add missing functionality as the need presents itself. This almost seductive feature of the paradigm has tempted many conscientious developers to turn lean, well-thought-out classes into huge dinosaurs that embody a tremendous amount of code—most of which is unused by the vast majority of its clients. Figure 0-2 illustrates what can happen when the functionality in a simple `String` class is allowed to grow to fill the needs of all clients. Each time a new feature is added for one client, it potentially costs all of the rest of the clients in terms of increased instance size, code size, runtime, and physical dependencies.

C++ programs are often larger than necessary. If care is not taken, the executable size for a C++ program could be much larger than it would be if the equivalent program were written in C. By ignoring external dependencies, overly ambitious class developers have created sophisticated classes that directly or indirectly depend on enormous amounts of code. A "Hello World" program employing one particularly elaborate `String` class produced an executable size of 1.2 megabytes!

```
// str.h
#ifndef INCLUDED_STR
#define INCLUDED_STR

class String {
    char *d_string_p;
    int d_length;
    int d_size;
    int d_count;
    // ...
    double d_creationTime;

  public:
    String();
    String(const String& s);
    String(const char *cp);
    String(const char c);
    // ...
    ~String();
    String &operator=(const String& s);
    String &operator+=(const String& s);
    // ...
    // (27 pages omitted!)
    // ...
    int isPalindrome() const;
    int isNameOfFamousActor() const;
};

// ...

#endif
```

```
// str.c
#include "str.h"
#include "sun.h"
#include "moon.h"
#include "stars.h"
// ...
// (lots of dependencies omitted)
// ...
#include "everyone.h"
#include "theirbrother.h"
String::String()
: d_string_p(0)
, d_length(0)
, d_size(0)
, d_count(0)
// ...
// ...
// ...
```

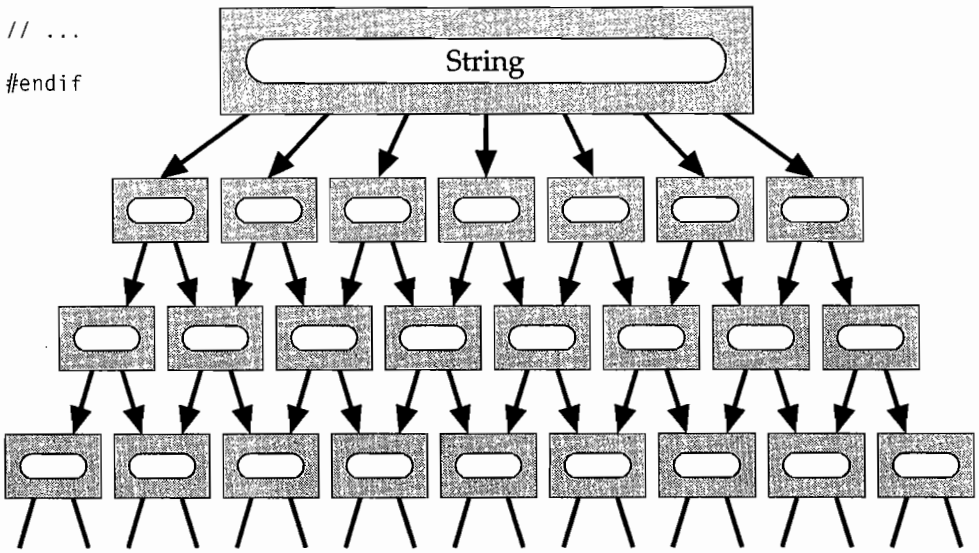


Figure 0-2: Oversized, Heavy-Weight, Non-Reusable `String` Class

Overweight types such as this `String` class not only increase executable size but can make the linking process unduly slow and painful. If the time necessary to link in `String` (along with all of its implementation dependencies) is large relative to the time it would otherwise take to link your subsystem, it becomes less likely that you would bother to reuse `String`.

Fortunately, techniques exist for avoiding these and other forms of unwanted link-time dependencies.

0.2.3 Excessive Compile-Time Dependencies

If you have ever tried to develop a multi-file program in C++, then you know that changing a header file can potentially cause several translation units to recompile. At the very early stages of system development, making a change that forces the entire system to recompile presents no significant burden. As you continue to develop your system, however, the idea of changing to a low-level header file becomes increasingly distasteful. Not only is the time necessary to recompile the entire system increasing, but so is the time to compile even individual translation units. Sooner or later, there comes a point where you simply refuse to modify a low-level class because of the cost of recompiling. If this sounds familiar, then you may have experienced excessive compile-time dependencies.

Excessive compile-time coupling, which is virtually irrelevant for small projects, can grow to dominate the development time for larger projects. Figure 0-3 shows a common example of what appears to be a good idea at first but turns bad as the size of a system grows. The `myerror` component defines a `struct`, `MyError`, that contains an enumeration of all possible error codes. Each new component that is added to the system naturally includes this header file. Unfortunately, each new component may have its own error codes that have not already been identified in the master list.

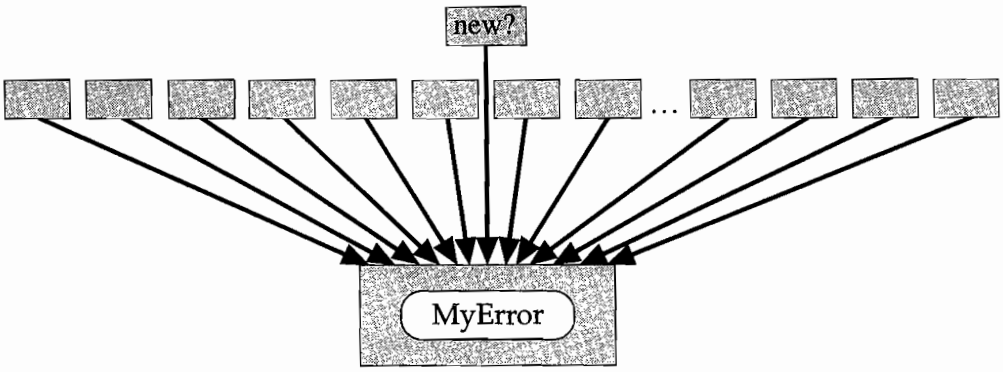
```
// myerror.h
#ifndef INCLUDED_MYERROR
#define INCLUDED_MYERROR

struct MyError {
    enum Codes {
        SUCCESS = 0,
        WARNING,
        ERROR,
        IO_ERROR,
        // ...
        READ_ERROR,
        WRITE_ERROR,
        // ...
        // ...
        BAD_STRING,
        BAD_FILENAME,
        // ...
        // ...
        CANNOT_CONNECT_TO_WORK_PHONE,
        CANNOT_CONNECT_TO_HOME_PHONE,
        // ...
        // ...
        MARTIANS_HAVE_LANDED,
        // ...
    };
};

#endif
```

Figure 0-3: An Insidious Source of Compile-Time Coupling

As the number of components gets larger, our desire to add to this list will wane. We will be tempted to reuse existing error codes that are, perhaps, only roughly appropriate just to avoid changing `myerror.h`. Eventually, we will abandon any thought of adding a new error code, and simply return `ERROR` or `WARNING` rather than change `myerror.h`. By the time we reach this point, the design is unmaintainable and practically useless.



There are many other causes of unwanted compile-time dependencies. A large C++ program tends to have many more header files than an equivalent C program. The unnecessary inclusion of one header file by another is a common source of excessive coupling in C++. In Figure 0-4, for example, it is not necessary to include the definition of objects in the simulator header file *just* because a client of class `Simulator` may find these definitions useful. Doing so forces the client to depend at compile-time on all such components whether or not they are actually used. Excessive include directives not only increase the cost of compiling the client, but increase the likelihood that the client will need to be recompiled as a result of a low-level change to the system.

By ignoring compile-time dependencies, it is possible to cause each translation unit in the system to include nearly every header file in the system, reducing compilation speed to a crawl. One of the first truly large C++ projects (literally thousands of staff years) was a CAD framework product developed at Mentor Graphics. The developers initially had no idea how much compile-time dependencies would impede their efforts. Even using our large network of workstations, recompiling the entire system was taking on the order of a week!

The problem was due to organizational details illustrated in part by the simulator component shown in Figure 0-4. Cosmetic techniques were developed to mitigate this problem, but the real solution came when the unnecessary compile-time dependencies were eliminated.

```
// simulator.h
#ifndef INCLUDED_SIMULATOR
#define INCLUDED_SIMULATOR
#include "cadtool.h"                    // required by "IsA" relationship
#include "myerror.h"                    // bad idea (see Section 6.9)
#include "circuitregistry.h"            // unnecessary compile-time dependency
#include "inputtable.h"                 // unnecessary compile-time dependency
#include "circuit.h"                    // required by "HasA" relationship
#include "rectangle.h"                  // unnecessary compile-time dependency
// ...
#include <iostream.h>                   // unnecessary compile-time dependency

class Simulator : public CadTool {      // mandatory compile-time dependency
    CircuitRegistry *d_circuitRegistry_p;
    InputTable& d_inputTable;
    Circuit d_currentCircuit;           // mandatory compile-time dependency
    // ...
  private:
    Simulator(const Simulator &);
    Simulator& operator=(const Simulator&);
  public:
    Simulator();
    ~Simulator();
    // ...
    MyError::Code readInput(istream& in, const char *name);
    MyError::Code writeOutput(ostream& out, const char *name);
    MyError::Code addCircuit(const Circuit& circuit);
    MyError::Code simulate(const char *outputName,
                           const char *inputName,
                           const char *circuitName);
    Rectangle window(const char *circuitName) const;
    // ...
};

#endif
```

Figure 0-4: Unnecessary Compile-Time Dependencies

As with link-time dependencies, there are several specific techniques available for eliminating compile-time dependencies.

0.2.4 The Global Name Space

If you have ever worked on a multi-person C++ project, then you know that software integration is a common forum for unwanted surprises. In particular, the proliferation of global identifiers can become problematic. One obvious danger is that these names can collide. The consequence is that the individually developed parts of the system

cannot be integrated without modification. For larger projects with hundreds of header files, it can be difficult even to find the declarations of a global name.

For example, I have used object libraries that have consisted of literally thousands of header files. I recall trying to find the definition of a type `TargetId` at file scope that looked like a class (but wasn't):

```
TargetId id;
```

I remember trying to "grep"[3] through all of the thousands of header files looking for the definition, only to receive a message to the effect that there were too many files. I wound up having to nest the grep command in a shell script that split up the header files based on the first letter in order to pare down the problem into 26 problems of manageable size. I eventually discovered that the "class" I was looking for was not a `class` at all. Nor was it a `struct` or a `union`! As illustrated in Figure 0.5, the type `TargetId`, it turned out, was actually a `typedef` declaration at file scope for an `int`!

```
// upd_system.h
#ifndef INCLUDED_UPD_SYSTEM
#define INCLUDED_UPD_SYSTEM

typedef int TargetId;          // bad idea!
class upd_System {
    // ...
  public:
    // ...
};

#endif
```

Figure 0-5: Unnecessary Global Name Space Pollution

The `typedef` had introduced a new type name into the global name space. There was no indication that type was an `int`, nor was there any hint of where I might find its definition.

[3] "grep" is a Unix search utility program.

```
// upd_system.h
#ifndef INCLUDED_UPD_SYSTEM
#define INCLUDED_UPD_SYSTEM

class upd_System {
    // ...
  public:
    typedef int TargetId;       // much better!
    // ...
};

#endif
```

Figure 0-6: Typedefs in Class Scope Are Easy to Find

Had the `typedef` declaration been nested within a class (as suggested in Figure 0-6), the reference would have been qualified with that class name (or the declaration would have been inherited), making it straightforward to track down:

```
upd_System::TargetId id;
```

Following simple practices like the one suggested above can minimize the likelihood of collisions and at the same time make logical entities easier to find in large systems.

0.2.5 Logical vs. Physical Design

Most books on C++ address only logical design. Logical design is that which pertains to language constructs such as classes, operators, functions, and so on. For example, whether a particular class should or should not have a copy constructor is a logical design issue. Deciding whether a particular operator (e.g., `operator==`) should be a class member or a free (i.e., nonmember) function is also a logical issue. Even selecting the types of the internal data members of a class would fall under the umbrella of logical design.

C++ supports an overwhelmingly rich set of logical design alternatives. For example, *inheritance* is an essential ingredient of the object-oriented paradigm. Another, called *layering*, involves composing types from more primitive objects, often embedded directly in the class definition. Unfortunately there are many who would try to use inheritance where layering is indicated: A `Telephone` is not a kind of `Receiver`, `Dial`, or `Cord`; rather, it is composed of (or "layered on") those primitive parts.

Misdiagnosing a situation in this way can lead to inefficiencies in both time and space, and can obscure the semantics of the architecture to a point where the entire system becomes difficult to maintain. Knowing when (and when not) to use a particular language construct is part of what makes the *experienced* C++ developer so valuable.

Logical design does not address issues such as where to place a class definition. From a purely logical perspective, all definitions at file scope exist at the same level in a single space without boundaries. Where a class is defined relative to its member definitions and supporting free operators is not relevant to logical design. All that is important is that these logical entities somehow come together to form a working program, and that, because the entire program is thought of as a single unit, there is no notion of individual *physical* dependencies. The program as a whole depends on itself.

There are several good books on logical design (see the bibliography). Unfortunately, there are also many problems, which arise only as programs get larger, that these books do not address. This is because much of the material relevant to successful large-system design falls under a different category, referred to in this book as *physical design.*

Physical design addresses the issues surrounding the physical entities of a system (e.g., files, directories, and libraries) as well as organizational issues such as compile-time or link-time dependencies between physical entities. For example, making a member `ring()` of class `Telephone` an `inline` function forces any client of `Telephone` to have seen not only the declaration of `ring()` but also its definition in order to compile. The logical behavior of `ring()` is the same whether or not `ring()` is declared `inline`. What is affected is the degree and character of the physical coupling between `Telephone` and its clients, and therefore the cost of maintaining any program using `Telephone`.

Good physical design, however, involves more than passively deciding how to partition the existing logical entities of a system. Physical design implications will often dictate the outcome of logical design decisions. For example, relationships between classes in the logical domain, such as IsA, HasA, and Uses, collapse into a single relationship, DependsOn, between components in the physical domain. Furthermore, the dependencies of a sound physical design will form a graph that has no cycles. Therefore we avoid logical design choices that would imply cyclic physical dependencies among components.

Simultaneously satisfying the constraints of both logical and physical design may, at times, prove challenging. In fact, some logical designs may have to be reworked or even replaced in order to meet the physical design quality criteria. In my experience, however, there have always been solutions that adequately address both domains, although it may (at first) take some time to discover them.

For small projects that fit easily into a single directory, physical design may warrant little concern. However, for larger projects the importance of a sound physical design grows rapidly. For very large projects, physical design will be a critical factor in determining the success of the project.

0.3 Reuse

Object-oriented design touts reuse as an incentive, yet like many other benefits of the paradigm, it is not without cost. Reuse implies coupling, and coupling in itself is undesirable. If several programmers are attempting to use the same standard component without demanding functional changes, the reuse is probably reasonable and justified.

Consider, however, the scenario where there are several clients working on different programs, and each is attempting to "reuse" a common component to achieve somewhat different purposes. If those otherwise independent clients are actively seeking enhancement support, they could find themselves at odds with one another as a result of the reuse: an enhancement for one client could disrupt the others. Worse, we could wind up with an overweight class (like the `String` class of Figure 0-2) that serves the needs of no one.

Reuse is often the right answer. But in order for a component or subsystem to be reused successfully, it must *not* be tied to a large block of unnecessary code. That is, it must be possible to reuse the part of the system that is needed without having to link in the rest of the system.

Not all code can be reusable. Attempting to implement excessive functionality or robust error checking for implementation objects can add unnecessarily to the development and maintenance cost as well as to the size of the executable.

Large projects stand to benefit from their implementors' knowing both when to reuse code and when to make code reusable.

0.4 Quality

Quality has many dimensions. *Reliability* addresses the traditional definition of quality (i.e., "Is it buggy?"). A product that is easy to use and does the right thing most of the time is often considered adequate. For some applications, however—in areas such as aerospace, medical, and financial, for example—errors can be extremely costly. In general, software cannot be made reliable through testing alone; by the time you are able to test it, the software's intrinsic quality has already been established. Not all software can be tested effectively. For software to be tested effectively, it must be designed from the start with that goal in mind.

Design for testability, although rarely the first concern of smaller projects, is of paramount importance when successfully architecting large and very large C++ systems. Testability, like quality itself, cannot be an afterthought: it must be considered from the start—before the first line of code is ever written.

There are many other aspects to quality besides reliability. *Functionality*, for example, addresses whether a product does what the customer expects. Sometimes a product will fail to gain acceptance because it does not have enough of the features that customers have come to expect. Worse, a product can miss its mark altogether: if a customer expects to buy a screwdriver, the best hammer in the world will fail a functionality test. Having a clear functional specification that meets marketing requirements *before* development is underway is an important first step toward ensuring appropriate functionality. In this book, however, we consider techniques that address how to design and build large systems, and not what large systems to design.

Usability is yet another measure of quality. Some software products can be very powerful in the right hands. However, it is not enough that the developer be able to use the product effectively. If the product is too complex, difficult, awkward, or painful for the typical intended customer to pick up and use, it will not be used. Often when we say *user*, we think of the *end user* of the system. In a large, hierarchically designed system, however, the clients of your component are probably just other components. Early feedback from customers (including other developers) is essential for ensuring usability.

Maintainability measures the relative cost to support a working system. Support includes such things as tracking down bugs, porting to new platforms, and extending the features of the product to meet the anticipated (or even unanticipated) future needs

of customers. A poorly designed system written in C++ (or any other language, for that matter) can be expensive to maintain and even more expensive to extend. Large, maintainable designs don't just happen; they are engineered by following a discipline that ensures maintainability.

Performance addresses how fast and small the product is. Although object-oriented design is known to have valuable advantages in the areas of extensibility and reuse, there are aspects of the paradigm that, if applied naively, can cause programs to run more slowly and require more memory than is necessary. If our code runs too slowly, or if it requires much more memory than a competitor's product, we cannot sell it. For example, modeling every character in a text editor as an object, although perhaps theoretically appealing, could be an inappropriate design decision if we are interested in optimal space/time performance.[4] Attempting to replace a heavily used fundamental type (such as `int`) with a user-defined version (such as a `BigInt` class) will inevitably degrade performance. If we fail to address our performance goals in the beginning, we may adopt architectures or coding practices that will preclude our ever achieving these goals, short of rewriting the entire system. Knowing where to accept some inelegance and knowing how to contain the effects of performance trade-offs distinguishes software engineers from mere programmers.

Each of these dimensions of quality is important to the overall success of a product. However, achieving each of them has one thing in common: we must consider each aspect of quality from the very start of a project. There is simply no way to add the quality once the design is complete.

0.4.1 Quality Assurance

Quality assurance (QA) is typically an organization within a company responsible for "assuring" that a certain measure of quality has been attained. A significant obstacle to achieving high-quality software is that QA often does not get involved until late in the development process, after the damage is already done. QA often does not influence the design of a software product. QA is rarely involved in low-level engineering design decisions. Typically, the testing that QA performs is at the end-user level, and it relies on the developers themselves for any low-level regression testing.

---

[4] See the Flyweight pattern in **gamma**, Chapter 4, pp. 195–206 for a clever solution to this particular kind of performance problem.

In this all-too-common process model, it is engineering's job to produce raw software that it then "throws over the wall" to QA. The software is often poorly documented, hard to understand, difficult to test, and unreliable. QA is now, somehow, expected to instill quality into the software. But how? Over and over, this model for assuring quality has demonstrated its ineffectiveness at achieving high-quality software in large projects. We now suggest a different model.

0.4.2 Quality Ensurance

QA must become an integral part of development. In this process model, developers have the responsibility for *ensuring* quality. That is, the quality must already be there in order for test engineers to find it.

In this process model, the distinction of QA and development is blurred; the technical qualifications for either position are essentially the same. One day, an engineer could write an interface and have another engineer review it for consistency, clarity, and usability. The next day the roles could be reversed. To be truly effective, the culture must be one of teamwork—each member helping the other to ensure high-quality software *as it is being developed.*

Providing a complete process model is a huge task and well beyond the scope of this book. However if high-quality software is to be achieved, system architects and software developers must take the lead by designing in the quality all along the way.

0.5 Software Development Tools

Large projects can benefit from many kinds of tools, including browsers, incremental linkers, and code generators. Even simple tools can be very useful. A detailed description of a simple dependency analyzer that I have found invaluable in my own work is provided in Appendix C.

Some tools can help to mitigate the symptoms of a poor design. Class browsers can help to analyze convoluted designs and find definitions for logical entities that would otherwise be hidden—buried within a large project. Sophisticated programming environments with incremental linkers and program databases can help to push the envelope of what can be accomplished even with a poor physical design. But none of these tools address the underlying problem: a lack of inherent design quality.

Unfortunately there is no single quick and easy way to achieve quality. Tools alone cannot solve fundamental problems resulting from a poor physical design. Although tools can postpone the onset of some of these symptoms, no tool will design in the quality for you, nor will it ensure that your design complies with its specification. Ultimately, it is experience, intelligence, and discipline that yield a quality product.

0.6 Summary

C++ is a whole lot more than just an extension of C. Cyclic link-time dependencies among translation units can undermine understanding, testing, and reuse. Unnecessary or excessive compile-time dependencies can increase compilation cost and destroy maintainability. A disorganized, undisciplined, or naive approach to C++ development will virtually guarantee that these problems occur as projects become larger.

Most C++ design books address only logical issues (such as classes, functions, and inheritance) and ignore physical issues (such as files, directories, and dependencies). In larger systems, however, physical design quality will dictate the correct outcome of many logical design decisions.

Reuse is not without cost. Reuse implies coupling, and coupling can be undesirable. Unwarranted reuse is to be avoided.

Quality has many dimensions: reliability, functionality, usability, maintainability, and performance. Each of these dimensions contributes to the success or failure of large projects.

Achieving quality is an engineering responsibility: it must be actively sought from the start. Quality is not something that can be added after a project is largely complete. For a QA organization to be effective, it must be an integral part of the entire design process.

Finally, good tools are an important part of the development process. But tools cannot make up for a lack of inherent design quality in large C++ systems. This book is about how to design in that quality.

PART I: BASICS

This book covers quite a bit of material relating to object-oriented design and C++ programming. Not all readers will have the same background. In Part I of this text, we address the fundamentals in an effort to reach a common starting point from which to launch further discussions.

Chapter 1 is a review of several key properties of the C++ language, basic object-oriented design principles and notation, and standard coding and documentation conventions used throughout this text. The purpose of this chapter is to help level the field. It is expected that much of this material will be familiar to many readers. Nothing presented here is new. Expert C++ programmers may choose to skim this chapter or simply refer to it as needed.

Chapter 2 describes a modest collection of commonsense design practices that most experienced software developers have already discovered. Adherence to the fundamental rules presented here is an integral part of successful software design. These rules also serve to frame the more advanced and subtle principles and guidelines presented throughout the book.