12 System Design

For large systems, design must take into account not only the performance of the finished product, but also its development and maintenance. Two anecdotes illustrate the problems of large-scale development.

1. Lakos worked at Mentor Graphics, a company that developed one of the first general-purpose graphics libraries for UNIX systems. Programmers were in the habit of `#include`ing all of the headers that they thought their component might need. The result was that compiling the library on a network of workstations required ***more than a week***.

   The source code was carefully reviewed and unnecessary `#include`s were removed. Compile time was significantly reduced (Lakos 1996).

2. A major software product was completed, the source code and executables were written to CDs, and the product was ready to be shipped to the customer. A couple of days later, a programmer made a small change to a member function, something like this: 473

```
class Widget
{
public:
   double get()
   {
       ++accessCount;             // this line added
       return size;
   }
private:
   static double size;
   unsigned long accessCount;
};
```

   After this change, the program crashed during most runs.

   The problem was solved after several days of debugging. The following code was the culprit: 474

```
Widget *pw;
// .... several pages of code
s = pw->get();
```

12.1 Logical and physical design

logical design

Object-oriented ***logical design*** concerns the organization of classes from the point of view of functionality, inheritance, layering, and so on. It is discussed in many books on C++ programming and software engineering, often with the use of UML diagrams, patterns, and other aids.

physical design

Physical design concerns the distribution of the various classes and other entities into files, directories, and libraries. Physical design is important for large projects but is less often discussed in books. Unfortunately, the only reference that explicitly covers physical design issues in industry-sized C++ projects is (Lakos 1996). "Unfortunate" because this book is now 20 years old and most of the technical aspects it describes have become outdated.

Logical and physical design are closely related. If the logical design is bad, it will not usually be possible to obtain a good physical design. But a good logical design can be spoilt by a poor physical design.

Logically, a system consists of classes (including enumerations) and free functions, with classes providing the main structure. 475

Physically, a system consists of ***components***. There are various ways of defining components but, fortunately, there is one precise definition that is suitable for systems written mostly or entirely in C++:

> ***A component consists of an implementation file (`.cpp`) and one or more header (`.h`) files.***

The header and implementation files may both use `#include` to read other header files; and parts of these may be skipped by conditional preprocessor directives.

compilation units

translation units

The compiler turns each component into an object file (`.obj`) or sometimes a library file (`.lib`/`.a`/`.o`/`.so` or (in Windows) `.dll`). The linker takes all the object and library files and creates an `executable`. Components are also called ***compilation units*** (see Section 1.2.1 on page 6) and ***translation units***.

Since C++ does not allow a class declaration to be split over several files, the definition of component implies that ***every class belongs to exactly one component***. However, the definition does allow a component to contain more than one class and, in fact, it is often useful to define components with several classes:

- Several closely-related classes may be declared in a single header file.
- A class, or classes, needed by a single component can be declared in the implementation file of that component.

In general, the ***logical structure*** of a design is determined by its classes; the ***physical structure*** of a design is determined by its components.

There are various kinds of logical association between two classes: a class can contain an instance of another class or a pointer to such an instance; inherit another class; have a value of another class passed to it by value, reference, or as a pointer; and so on. Physical associations are, in general, simpler: one component either uses another component or it doesn't.

A ***client*** of component C is another component of the system that uses C. With a few exceptions (see Section 12.5.2 on page 261), clients can be recognized because they `#include` header files: component `C1` is a client of `C2` if either `C1.h` or `C1.cpp` needs to read `C2.h`. Note that: 476

- "Reading" `C2.h` is not the same thing as `#include`ing `C2.h` because a client might `#include` `X.h` which `#include`s `C2.h`. In Figure 157, compiling `C1.cpp` requires reading `C2.h`.
- The client relationship is transitive: if `C1` is a client of `C2`, and `C2` is a client of `C3`, then `C1` is a client of `C3`.

```
#ifndef C1_H                    #ifndef X_H
#define C1_H                    #define X_H

#include "X.h"                  #include "C2.h"
....                            ....
#endif                          #endif
```

Figure 157: Physical dependency: `c1.h` "reads" `c2.h`

12.2 Linkage

We discussed linkage issues previously (see Section 4.3 on page 72). To get good physical structure, we want to minimize dependencies between components. Doing this requires knowing how dependencies arise.

Declarations in a header or implementation file have no effect outside the file (except, of course, that declarations in a header file are read in the corresponding implementation file). 477

Definitions in a header or implementation file cause information to be written to the object file, and therefore create dependencies. As we have seen, definitions should be avoided altogether in header files.

General rules:

- Put only constant, enumeration, and class declarations into header files. 478
- Put all data and function definitions into implementation files.
- Put data inside classes wherever possible.
- If you have to put data at file scope, declare it as `static`.
- Declarations ***and*** definitions for template classes must be put into header files, so that the compiler can instantiate templates.

The compiler considers inlined functions to be declarations, not definitions, even if they have bodies. It is therefore acceptable to put, for example, 479

```
inline double sqr(double x) { return x * x; }
```

in a header file. However, if you remove the `inline` qualifier, the program may not compile! This is because the header file may be read more than once, putting two copies of the function `sqr` into the program, which causes problems for the linker (we discuss inlining in more detail in Section 12.5.5 on page 266).

12.3 Namespaces

Namespaces in C++ are used similar to packages in Java. They are helpful in structuring large-scale projects and keeping them manageable. To declare your own namespace `foospace`, you'd write:

namespace

480

```
namespace foospace
{
   class foo
   {
       ....
   };
   ....
}
```

Inside the namespace declarations, you put your own classes, functions, etc. For example, the class `foo` defined inside the namespace above is now referenced as `foospace::foo`.

nested namespaces

The `using` directive for your own namespaces works just as for the standard namespace we've seen so far. You can also have nested namespaces, like in `foospace::barspace`. A class `bar` declared in the inner namespace would then be referenced as `foospace::barspace::bar`.

default namespace and `::` prefix

All classes defined outside a namespace declaration go into the ***default namespace*** and can be referenced with a `::` prefix. For example, if you have a class `baz` in the default namespace, you can reference it with `::baz`. This might be necessary in case there is also a `baz` in another namespace you are `using` in your code.

anonymous namespace

Finally, you can have an unnamed, ***anonymous namespace***. Everything inside the anonymous namespace becomes invisible outside the translation unit. Thus, in term of linking, they are a (better) alternative to `static` variables at global scope (we will discuss global variables in more detail below). Note that this is the only way to control visibility: there is no C++ equivalent to package visibility in Java. For more on namespaces, see (Weiss 2004, Chapter 4.15), (Dewhurst 2005, Item 23), (Prata 2012, pp.482–497), and (Stroustrup 2013, Chapter 14).

12.4 Cohesion

Every book on Software Engineering includes the slogan ***low coupling, high cohesion***. This mantra is useful only if we know:

- what is coupling?
- what is cohesion?
- how do we reduce coupling?

- how do we increase cohesion?

Coupling is probably the more important of these two aspects of a system, and we discuss it in Section 12.5 on the next page.

Cohesion

Cohesion is the less important partner of the "low coupling, high cohesion" slogan. Roughly speaking, a component is ***cohesive*** if it is independent of other components (as far as possible) and performs a single, well-defined role, or perhaps a small number of closely related tasks. A component is not cohesive if its role is hard to define or if it performs a variety of loosely related tasks.

One way to define cohesion is by saying that the capabilities of a class should be necessary and sufficient (like a condition in logic). The capabilities of a component are ***necessary*** if the system cannot manage without it. The capabilities of a class are ***sufficient*** if they jointly perform all the services that the client requires. 481

Necessary	=	every capability provided is in fact required by the system
	=	***no unused capabilities***
Sufficient	=	every capability required by the system is in fact provided
	=	***no missing capabilities***

The "necessary and sufficient" criterion is not the whole story. For example, we could write the entire system as one gigantic class that provided the required functionality and had no superfluous functions; clearly, this class would be both necessary and sufficient, but it would also be unmanageable and probably useless.

Sufficiency is a precise concept, but there is some leeway. For example, if it turns out that the sequence 482

```
p->f();
p->g();
```

occurs often in system code – perhaps, in fact, ***every*** invocation of `f` is followed by a call to `g` – then it probably makes sense to add a function `h` that combines `f` and `g` to the class. The old class was technically sufficient, but the new function improves clarity, simplifies maintenance, and may even make the program run a little faster.

Clearly there is no point in taking the time to code and test functions that will never be used. Nevertheless, there are programmers who like to spend large amounts of time writing code that "might be useful one day" rather than focusing on writing or improving code that was actually needed yesterday.

Consequently, cohesion also implies a division of the system into components of manageable size. A good guideline is that it should be possible to describe the role of a component with a single sentence. If you ask what a component does and the answer is either a mumble or a 10 minute peroration, there is probably something wrong with the design of the component.

Cohesion is related to coupling in the following way: a component that tries to do too much (that is, plays several roles) is likely to have many clients (perhaps one or more for each role). It may also need to be a client of several other components in order to perform its roles. Consequently, its coupling is likely to be high. On the other hand, a cohesive component is likely to have few clients and few dependencies, and therefore lower coupling.

There are always exceptions.

- A highly-cohesive component might provide an essential service to many parts of a system; in this case it would contribute to heavy coupling.
- Low cohesion in the design can sometimes be corrected by careful physical design. For example, a group of related classes could be put into a single component, exposing only some class interfaces to the rest of the system.

12.5 Coupling

***Coupling* is any form of dependency between components.**

483

Coupling is not an all-or-nothing phenomenon: there are ***degrees*** of coupling. At one end of the scale, if a component has no coupling with the rest of the system, it cannot be doing anything useful and should be thrown away. It follows that ***some*** coupling is essential and therefore that the issue is ***reducing*** coupling, not eliminating it.

The other end of the scale is very high coupling: every component of the system is strongly coupled to every other component. Such a system will be very hard to maintain, because a change to one component often requires changes to other components. In the following, we look at different techniques for reducing coupling.

12.5.1 Encapsulation

The first step towards low coupling is ***encapsulation***, which means hiding the implementation
484 details of an object and exposing only a well-defined public interface. Figure 158 is a first (rather feeble) attempt at defining a class for points with two coordinates.

```
class Point1
{
public:
    double x;
    double y;
};
```

Figure 158: A class for points: version 1

Class `Point1` provides very poor encapsulation: anyone can get and set its coordinates, and it has no control over its state at all. Obviously, we should make the coordinate data `private`, but
485 then we would have to provide some access functions, as shown in Figure 159 on the next page.

Access functions that provide no checking, like these, are not much better than `public` data. But access functions do at least provide the ***possibility*** of controlling state and maintaining
486 class invariants. For example, `setX` might be redefined as

```
class Point2
{
public:
    double getX() { return x; }
    double getY() { return y; }
    void setX(double nx) { x = nx; }
    void setY(double ny) { y = ny; }
private:
    double x;
    double y;
};
```

Figure 159: A class for points: version 2

```
void Point2::setX(double nx)
{
    if (nx < X_MIN) nx = X_MIN;
    if (nx > X_MAX) nx = X_MAX;
    x = nx;
}
```

Access functions provide a way of hiding the representation of an object. For example, Figure 160 shows how we could define points using complex numbers without affecting users in any way (except, perhaps, efficiency). 487

```
class Point3
{
public:
    double getX() { return z.re; }
    double getY() { return z.im; }
    void setX(double nx) { z.re = nx; }
    void setY(double ny) { z.im = ny; }
private:
    complex<double> z;
};
```

Figure 160: A class for points: version 3

Even with access functions, however, `Point` hardly qualifies as an object. Why do users want points? What are the operations that points should provide? A class for points should look more like Figure 161 on the following page, in which function declarations appear in the class declaration, but function definitions are in a separate implementation file. 488

`Point4` has less coupling than `Point3` in another respect. If any of the inline functions of `Point3` are changed, its clients will have to be recompiled. Since the functions of `Point4` are defined in `Point4.cpp` rather than `Point4.h`, they can be changed without affecting clients (although the system will have to be re-linked, of course).

```
class Point4
{
public:
    void draw();
    void move(double dx, double dy);
    ....
private:
    // Hidden representation
};
```

Figure 161: A class for points: version 4

A client of one of the `Point`n classes will be coupled to the `Point`n component. The degree of coupling depends on the `Point` class: it is highest for `Point1` (the user has full access to the coordinates) and lowest for `Point4` (the user can manipulate points only through functions such as `move` and `draw`).

From the point of view of the owner of the `Point` class, lower coupling means greater freedom. If the owner of `Point1` makes almost any change at all, all clients will be affected. The owner of `Point4`, on the other hand, can change the representation of a point, or the definitions of the member functions, without clients needing to know, provided only that the class continues to meet its specification. This is one advantage of low coupling:

Low coupling makes maintenance easier.

A complicated object is entitled to have a few access functions but, in general, a class should ***keep a secret***.[67] If your class seems to need a lot of `get` and `set` functions, you should seriously consider redesigning it.

Here are formal definitions (Lakos 1996, pages 105 and 138):

1. The ***logical interface*** of a component is the part that is programmatically accessible or detectable by a client. 489
2. An implementation entity (type, variable, function, etc.) that is not accessible or detectable programmatically through the logical interface of a component is ***encapsulated*** by that component.

encapsulation

In set theoretic notation, for any component C

$$I_C \cup E_C = U \qquad \text{and} \qquad I_C \cap E_C = \emptyset$$

where I_C is the set of implementation entities in the logical interface of C, E_C is the set of entities encapsulated by C, and U is the "universe" of all entities in C.

[67]The useful metaphor "keeping a secret" was introduced by David Parnas in a very influential paper (Parnas 1978).

12.5.2 Hidden coupling

C++ provides various ways in which coupling between components can be hidden: that is, there is a dependency between two components even though neither #includes the other's header file. For example, the following components A and B are coupled: 490

```
// file A.cpp                         // file B.cpp

int numWidgets;                       extern int numWidgets;

....                                  ....
```

Using the definitions of the previous section, we note that numWidgets is in the logical interface of component A, because it can be accessed by component B by using extern. extern

The object file obtained when A.cpp is compiled will allocate memory for numWidgets. The object file obtained when B.cpp is compiled will not allocate memory for numWidgets but will expect the linker to provide an address for numWidgets. If the definition in A.cpp is changed or removed, both components will compile, but the system won't link.

Don't use global variables.

Don't use extern ***declarations.***

12.5.3 Compilation dependencies

Coupling can affect the time needed to rebuild a system. Build times are significant for large projects, and it is useful to know how to keep them low. If we define class Point5 as in Figure 162, 491 sizeof(Point5) gives 16 bytes, showing that the compiler allocates two 8-byte double values to store a Point5. If we decide to add a new data member, d, to store the distance of the point from the origin, we obtain Point6, shown in Figure 163 on the next page. Then sizeof(Point6) gives 492 24 bytes, showing that the compiler allocates three 8-byte double values to store a Point6.

```
class Point5
{
private:
    double x;
    double y;
};
```

Figure 162: A class for points: version 5

At this point, we might decide that our class needs a function, as shown in Figure 164. Adding a 493 function has ***no effect*** on the size of the class: sizeof(Point7) is 24 bytes, just like Point6.

But if we make the function virtual, as in Figure 165 on the following page, then sizeof(Point8) 494

```
class Point6
{
private:
    double x;
    double y;
    double d;
};
```

Figure 163: A class for points: version 6

```
class Point7
{
public:
    void move(double dx, double dy) { x += dx; y += dy; }
private:
    double x;
    double y;
    double d;
};
```

Figure 164: A class for points: version 7

gives 32 bytes, because a virtual function requires a ***virtual function table*** or ***vtable*** for short.[68]

```
class Point8
{
public:
    virtual void move(double dx, double dy) { x += dx; y += dy; }
private:
    double x;
    double y;
    double d;
};
```

Figure 165: A class for points: version 8

In summary, adding or removing data members changes the size of the objects. Adding a virtual function also changes the size. (Adding more virtual functions after the first makes no difference, because there is only one vtable and it contains the addresses of all virtual functions.)

The changes we have been discussing should not affect users of the class. Private data members cannot be accessed anyway, and whether a function is virtual affects only derived classes. Nevertheless, if we change the size of a `point`, ***every component that includes*** `point.h`

[68]The pointer to the vtable needs 4 bytes on a 32-bit Intel architecture. The additional 4 bytes may be added for alignment purposes.

will be recompiled during the next build! Since building a large project can take hours or even days, changing a class declaration, even the `private` part, is something best avoided (see Figure 166[69]).

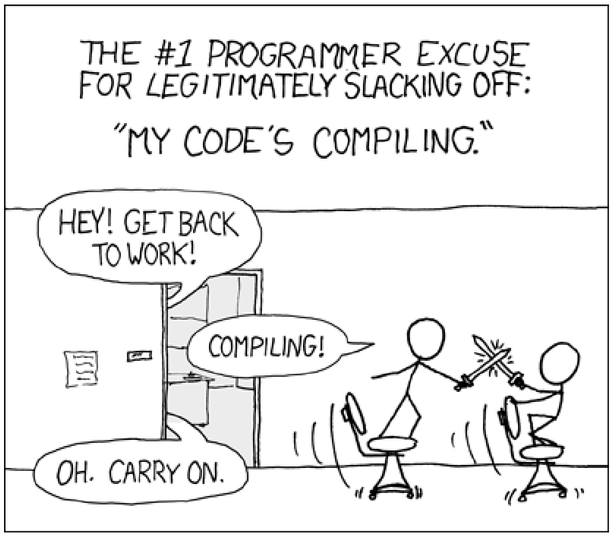

Figure 166: The importance of reducing compile time

To see why this happens, consider the first program in Figure 167. To allocate stack space for `p`, the compiler must be able to compute `sizeof(Point)`. To compute this, it must see the declaration of class `Point`. Finally, to see the declaration, it must read `"point.h"`. If `"point.h"` changes, the program must be recompiled. 495

```
#include "point.h"

int main()
{
    Point p;
    ....
}
```

Figure 167: Forward declarations: 1

The program in Figure 168 on the following page is slightly different. The compiler does ***not*** need to know `sizeof(Point)` to compile this program, because all pointers and references have the same size (the size of an address: usually 4/8 bytes on a 32bit/64bit system). Consequently, reading the declaration of class `Point` is a waste of time. However, the compiler ***does*** need to know that `Point` is a class and, for this purpose, the forward declaration of Figure 169 is sufficient. 496

forward declarations

497

Don't include a header file when a forward declaration is all you need.

[69]Copyright Randall Munroe, http://xkcd.com/303/, licensed under a Creative Commons Attribution-NonCommercial 2.5 License

```
#include "point.h"

int main()
{
    Point* pp;
    ....
    void f(Point & p);
    ....
}
```

Figure 168: Forward declarations: 2

```
class Point; // forward declaration

int main()
{
    Point* pp;
    ....
    void f(Point & p);
    ....
}
```

Figure 169: Forward declarations: 3

498 In early versions of the standard libraries, names such as `ostream` referred to actual classes. Consequently, the code shown in Figure 170 on the next page worked well in a header file. Times have changed, and `ostream` is now defined by `typedef`. However, the standard library defines a header file that contains forward declarations for all stream classes: see Figure 171 on the facing page.

499 Suppose that points have colours. It is tempting to put the enumeration declaration outside the class declaration:

```
enum Colour { RED, GREEN, BLUE };

class Point
{
    ....
```

This is convenient, because we can use the identifiers `RED`, `GREEN`, and `BLUE` wherever we like. But this convenience is also a serious drawback: the global namespace is ***polluted*** by the presence of these new names. It is much better to put the enumeration ***inside*** the class declaration, like this:

```
class Point
{
public:
    enum Colour { RED, GREEN, BLUE };
    ....
```

```
class ostream;
....
friend ostream & operator<<(ostream & os, const Widget & w);
```

Figure 170: Forward declaration for `ostream`: 1

```
#include <iosfwd>
....
friend ostream & operator<<(ostream & os, const Widget & w);
```

Figure 171: Forward declaration for `ostream`: 2

We now have to write `Point::RED` instead of `RED`, but there is no longer any danger of the colour names clashing with colour names introduced by someone else.

In situations where class declarations alone are inadequate, for example in the development of a library, we can use namespaces instead (which are discussed in Section 12.3 on page 256).

12.5.4 Cyclic Dependencies

The worst kind of dependencies are ***cyclic dependencies***. When a system has cyclic dependencies, "nothing works until everything works". Some cyclic dependencies are unavoidable, but many can be eliminated by careful design.

Suppose we have classes `Foo` and `Bar` that are similar enough to be compared. We might write 500

```
class Foo
{
public:
    operator==(const Bar & b);
    ....
};

class Bar
{
public:
    operator==(const Foo & f);
    ....
};
```

This creates a cyclic dependency between `Foo` and `Bar`. The dependency can be eliminated by using `friend` functions and forward declarations:

In file `foo.h`: 501

```
class Bar;

class Foo
{
    friend operator==(const Foo & f, const Bar & b);
    friend operator==(const Bar & b, const Foo & f);
    ....
};
```

502 In file `bar.h`:

```
class Foo;

class Bar
{
    friend operator==(const Foo & f, const Bar & b);
    friend operator==(const Bar & b, const Foo & f);
    ....
};
```

503 In an implementation file (`foo.cpp`, or `bar.cpp`, or somewhere else):

```
operator==(const Foo & f, const Bar & b) { .... }
operator==(const Bar & b, const Foo & f) { .... }
```

12.5.5 Inlining

Normally, a function is called by evaluating its arguments, placing them in registers or on the stack, storing a return link in the stack, and passing control to the function's code. When the function returns, it uses the return link to transfer control back to the call site. There is clearly a fair amount of overhead, especially if the function is doing something trivial, such as returning the value of a data member of a class.

If the compiler has access to the definition of a function, it can compile the body of the function directly, without the call and return. This is called ***inlining*** the function.

An inline function will usually be faster than a called function, although the difference will be significant only for small functions. (For large functions, especially if they have loops or recursion, the time taken by calling and returning will be negligible compared to the time taken to execute the function.)

Heavy use of inlined functions increases the size of the code, because the code of the function is compiled many times rather than once only. Thus inlining is an example of a ***time/space tradeoff***.

Inlining is relevant to this section because ***inlining affects coupling***. The compiler can inline a function only if its definition is visible. Within an implementation file, a definition of the form

```
inline void f() { .... }
```

permits the compiler to inline `f`, but only in that particular implementation file. To inline a function throughout the system, the `inline` qualifier, and the body of the function, must appear in a header file. As we have seen, this implies that any change to the function body will force recompilation of all clients.

```
#ifndef WIDGET_H
#define WIDGET_H

class Widget
{
public:
    int f1() { return counter; }
    int f2();
private:
    int counter;
};

inline double f3()
{
    ....
}

char f4();
```

Figure 172: Inlining

Whenever a function definition appears inside a class declaration, the compiler is allowed to compile it inline. In Figure 172, the compiler may inline `f1` and `f3` and it cannot inline `f2` and `f4`. 504

The qualifier `inline` is a ***compiler hint***, not a directive. The compiler is not obliged to inline functions that are defined inside the class declaration or declared `inline`, and it may inline functions of its own accord.

Do not place too much dependence on inlining: the rewards are not great and there may be drawbacks. Herb Sutter (Sutter 2002, pages 83–86) argues that good compilers know when and when not to inline and programmers should leave the choice to them. Scott Meyers (Meyers 2005, Item 30) discusses inlining in more detail.

12.6 The Pimpl Idiom

A complicated class might have quite a large `private` part. Every time the owner of the class changes the class declaration, every client gets recompiled. A "Pimpl" is a simple way of avoiding this dependence: it is a mnemonic for ***pointer to implementation***.

We will use a slightly modified version of the class `Account` from Section 6 on page 99 to illustrate the Pimpl idiom. We will develop three versions of this class, requiring each one of them to execute the test program shown in Figure 173 on the following page, giving the results 505

506

```
#include "account.h"

Account cbv(Account val)
{
   val.withdraw(2000);
   cout << val << endl;
   return val;
}

void cbr(Account & val)
{
   val.withdraw(2000);
   cout << val << endl;
}

int main()
{
    Account acc("Fred", 10000);
    cout << acc << endl;
    acc.deposit(2000);
    cout << acc << endl;
    acc.withdraw(3000);
    cout << acc << endl;
    cout << "Assign\n";
    Account cop = acc;
    cout << "CBV\n";
    cbv(acc);
    cout << "CBR\n";
    cbr(cop);
    return 0;
}
```

Figure 173: Test program for class `Account`

of Figure 174.

507 The ID numbers in the output show that passing by value and returning by value cause new copies of the account to be created, but passing by reference does not cause any creation.

508 509 Figure 175 on the next page shows the header file of the original class `Account`. There are a few small differences between this version and the earlier version:

- We use a `std::string` to represent the name of the account (like in the version used in the inheritance lecture).

- Unique ID numbers are generated automatically using a `static` counter.

510 511 512 513 Figures 176 and 177 show the implementation of class `Account`. Note that each constructor

```
Fred        1  100.00
Fred        1  120.00
Fred        1   90.00
Assign
CBV
Fred        3   70.00
CBR
Fred        2   70.00
```

Figure 174: Output from the program of Figure 174

```
class Account
{
public:
   Account();
   Account(std::string name, long balance = 0);
   Account(const Account & other);
   Account & operator=(const Account & other);
   const std::string getName() const;
   void deposit(long amount);
   void withdraw(long amount);
   void transfer(Account & other, long amount);
   friend std::ostream & operator<<(std::ostream & os, const Account & acc);

private:
   static long idGen;
   long id;
   std::string name;
   long balance;
};

bool operator==(const Account & left, const Account & right);
bool operator!=(const Account & left, const Account & right);
```

Figure 175: Header file for original class `Account`

increments the account ID. Assignment also increments the ID and the insert `operator<<` displays this value.

The definitions for the comparison operators, `operator==` and `operator!=`, will not be shown again because they do not change.

12.6.1 Introducing Pimpl

Converting this class to a Pimpl class requires changing the head and implementation files. Figure 178 on page 272 shows the new header file. Two changes are required: 514

```
long Account::idGen = 0;

Account::Account()
      : id(++idGen), name(""), balance(0) {}

Account::Account(string name, long balance)
      : id(++idGen), name(name), balance(balance) {}

Account::Account(const Account & other)
      : id(++idGen), name(other.name), balance(other.balance) {}

Account & Account::operator= (const Account & other)
{
   if (this == &other)
      return *this;
   id = ++idGen;
   name = other.name;
   balance = other.balance;
   return *this;
}

const string Account::getName() const
{
   return name;
}

void Account::deposit(long amount)
{
   balance += amount;
}

void Account::withdraw(long amount)
{
   if (amount > balance)
      cerr << "Withdrawal greater than balance.\n";
   else
      balance -= amount;
}

void Account::transfer(Account & other, long amount)
{
   withdraw(amount);
   other.deposit(amount);
}
```

Figure 176: Implementation for class `Account`: part 1

```
ostream & operator<<(ostream & os, const Account & acc)
{
   return os <<
          left << setw(8) << acc.name << right <<
          setw(4) << acc.id <<
          fixed << setprecision(2) << setw(8) << acc.balance / 100.0;
}

bool operator==(const Account & left, const Account & right)
{
   return &left == &right;
}

bool operator!=(const Account & left, const Account & right)
{
   return !(left == right);
}
```

Figure 177: Implementation for class `Account`: part 2

- The implementation class `AccImpl` is declared before class `Account`.
- The private data of class `Account` is replaced by a pointer to an instance of `AccImpl`.

It would be nice to keep the existence of class `AccImpl` private, but the compiler gives errors if the declaration is moved to the `private` section of `Account`.

Figures 179, 180, and 181 show the new implementation file for `Account`.It has two parts: the declaration for class `AccImpl`, and the new function definitions for class `Account`. 515 516

Class `AccImpl` is very similar to the original class `Account`. It has the same private data and the same functions. We have added messages in the constructors and destructor to show that these functions are called in the correct sequence. In the hope of getting back some of the efficiency that we have lost by Pimpl'ing, all of the functions are inlined. 517 518 519

Insertion (`operator<<`) expects a pointer to an instance of `AccImpl`.

The next step is to define new versions of the functions of `Account`, as shown in Figure 181 on page 275. The constructors create a new instance of `AccImpl` and initialize it appropriately. The copy constructor is inefficient: it creates a default `AccImpl` and then copies information from the other `Account` object into it. The destructor is required because the implementation object is created dynamically and must be destroyed. 520 521 522 523

Assignment (`operator=`) is implemented in the same way as the copy constructor.

The remaining functions simply forward the original request to the implementation: function `f()` is implemented as `pimpl->f()`.

Insertion (`operator<<`) passes the Pimpl pointer to the version of `operator<<` defined for `AccImpl`.

Figure 182 on page 275 shows the output generated by the test program with the Pimpl'ed class. 524

```
class AccImpl;

class Account
{
public:
   Account();
   Account(std::string name, long balance = 0);
   ~Account();
   Account(const Account & other);
   Account & operator=(const Account & other);
   const std::string getName() const;
   void deposit(long amount);
   void withdraw(long amount);
   void transfer(Account & other, long amount);
   friend std::ostream & operator<<(std::ostream & os, const Account & acc);

private:
   AccImpl *pimpl;
};
```

Figure 178: Header file for Pimpl class `Account`

The additional lines show that the `AccImpl` objects are being created and destroyed correctly.

12.6.2 Combining Pimpl and autopointers

As a variation on the Pimpl idiom, we can use an autopointer instead of a normal pointer.
525 Figure 183 on page 276 shows the revised header for class `Account`.

The implementation of `AccImpl` does not change, so we do not show it. Nor do we show the unchanged functions of class `Account`. Figure 184 on page 277 shows the functions that are
526 changed. The constructors and `operator=` create instances of `AccImpl` and set autopointers
527 to them. The destructor is retained, but only because it displays a message: it is no longer necessary.

As expected, the output generated when the autopointer version is executed is the same as the output generated by the pointer version (Figure 182 on page 275).

The Pimpl idiom trades performance for compilation speed. The program executes slightly more slowly, because functions are called indirectly, but compiles faster.

The improvement in compilation time can be dramatic. Herb Sutter (Sutter 2000, page 110) says:

> *"I have worked on projects in which converting just a few widely-used classes to use Pimples has halved the system's build time."*

The ***Bridge design pattern*** (Section 13.5 on page 299) is an extension of the Pimpl idea. For more details on Pimpl, in particular using C++11 features, see (Meyers 2014, Item 22).

```
#include "account.h"

class AccImpl
{
public:
   AccImpl() : name(""), id(++idGen), balance(0)
   {
      cerr << "create " << id << endl;
   }

   AccImpl(std::string name, long balance = 0)
         : name(name), id(++idGen), balance(balance)
   {
      cerr << "create " << id << endl;
   }

   ~AccImpl()
   {
      cerr << "delete " << id << endl;
   }

   AccImpl & operator=(const AccImpl & other)
   {
      name = other.name;
      balance = other.balance;
   }

   const std::string getName() const
   {
      return name;
   }

   void deposit(long amount)
   {
      balance += amount;
   }

   void withdraw(long amount)
   {
      if (amount > balance)
         cerr << "Withdrawal amount greater than balance.\n";
      else
         balance -= amount;
   }

   void transfer(AccImpl & other, long amount)
   {
      withdraw(amount);
      other.deposit(amount);
   }
```

Figure 179: Implementation file for Pimpl class `Account`: part 1

```
   friend ostream & operator<<(ostream & os, AccImpl *p)
   {
      return os <<
             left << setw(8) << p->name << right <<
             setw(4) << p->id <<
             fixed << setprecision(2) << setw(8) << p->balance / 100.0;
   }

private:
   static long idGen;
   std::string name;
   long id;
   long balance;
};

long AccImpl::idGen = 0;

Account::Account()
{
   pimpl = new AccImpl();
}

Account::Account(std::string name, long balance)
{
   pimpl = new AccImpl(name, balance);
}

Account::Account(const Account & other)
{
   pimpl = new AccImpl();
   *pimpl = *(other.pimpl);
}

Account::~Account()
{
   delete pimpl;
}

Account & Account::operator=(const Account & other)
{
   if( this == &other )
       return *this;
   delete pimpl;
   pimpl = new AccImpl();
   *pimpl = *(other.pimpl);
   return *this;
}
```

Figure 180: Implementation file for Pimpl class `Account`: part 2

```
const std::string Account::getName() const
{
   return pimpl->getName();
}

void Account::deposit(long amount)
{
   pimpl->deposit(amount);
}

void Account::withdraw(long amount)
{
   pimpl->withdraw(amount);
}

void Account::transfer(Account & other, long amount)
{
   pimpl->transfer(other.pimpl, amount);
}

ostream & operator<<(ostream & os, const Account & acc)
{
   return os << acc.pimpl;
}
```

Figure 181: Implementation file for Pimpl class `Account`: part 3

```
create 1
Fred       1  100.00
Fred       1  120.00
Fred       1   90.00
Assign
create 2
CBV
create 3
Fred       3   70.00
create 4
delete 4
delete 3
CBR
Fred       2   70.00
delete 2
delete 1
```

Figure 182: Output from the Pimpl version of class `Account`

```
#include <memory>

class AccImpl;

class Account
{
public:
   Account();
   Account(std::string name, long balance = 0);
   Account(const Account & other);
   ~Account();
   Account & operator=(const Account & other);
   const std::string getName() const;
   void deposit(long amount);
   void withdraw(long amount);
   void transfer(Account & other, long amount);
   friend std::ostream & operator<<(std::ostream & os, const Account & acc);

private:
   std::auto_ptr<AccImpl> pimpl;
};
```

Figure 183: Header file for autopointer Pimpl class `Account`

12.7 Refactoring

When we change a system to improve its maintainability, performance, or other characteristics ***without changing its functionality***, we are ***refactoring*** it. A common reason for refactoring is to increase cohesion and decrease coupling. Refactoring is a large topic and entire books have been written about it – for example, (Fowler, Beck, Brant, Opdyke, and Roberts 1999; Kerievsky 2004).[70]

> *"... merciless refactoring of existing code ... greatly decreases the sort of chaos I've seen in "clean and simple" code. It's almost like an emergent property, and I can't quite explain what's going on. The code becomes more fluid. The chunks are smaller so they have less trouble moving them to where they ought to be...*
>
> *Like design patterns, refactoring codifies wisdom. This wisdom is about what good code looks like. I've encountered most of the refactorings in my professional life, but that was over the course of many years. I envy programmers starting out today with this sort of wisdom at their fingertips. Established programmers like me didn't even realize this was something that needed codifying."* (Eric Hodges[71])

Here are brief sketches of some simple refactorings.

[70]See also the Refactoring Home Page, http://www.refactoring.com.
[71]See http://ootips.org/refactoring.html

```
Account::Account()
{
   pimpl = auto_ptr<AccImpl>(new AccImpl());
}

Account::Account(std::string name, long balance)
{
   pimpl = auto_ptr<AccImpl>(new AccImpl(name, balance));
}

Account::Account(const Account & other)
{
   pimpl = auto_ptr<AccImpl>(new AccImpl());
   *pimpl = *(other.pimpl);
}

Account::~Account()
{
   cerr << "delete " << id << endl;
}

Account & Account::operator=(const Account & other)
{
   if( this != &other ) {
       pimpl = auto_ptr<AccImpl>(new AccImpl());
       *pimpl = *(other.pimpl);
   }
   return *this;
}
```

Figure 184: Implementation file for autopointer Pimpl class `Account`

12.7.1 Simplify Calling Patterns

Browsing through source code (the first essential step for ***any*** refactoring), you notice these two lines: 528

```
double xFactor = minPlay(x, y);
target.setParams(xFactor, 997.3);
```

Looking further, you notice that these lines occur frequently in the code, with slight variations in parameters.

You could refactor the code by ***wrapping*** this pair into a single function:

```
void setp(Object target, double x, double y, double scale)
{
   double xFactor = minPlay(x, y);
   target.setParams(xFactor, scale);
}
```

It might be even better to add a new member function to class `Object` and then replace each pair of lines by

```
target.revisedSetParams(x, y, scale);
```

12.7.2 Introduce Design Patterns

Knowing design patterns helps refactoring. Browsing source code, you notice a group of three classes, `A`, `B`, and `C` that are tightly coupled (e.g., each class contains pointers to the others and uses their methods). Uses of these classes elsewhere in the code is complex and confusing: 529

```
pa->f();      // May change B and C
pb->g();      // May change A
....
```

Façade

Using the ***Façade*** pattern (Gamma, Helm, Johnson, and Vlissides 1995), you could create a fourth class, `D`, that provides an interface to the three classes `A`, `B`, and `C`. Class `D` might look something like this:

```
class D
{
public:
   D(A* pa, B* pb, C* pc) : pa(pa), pb(pb), pc(pc) : {}
   void f() { pa->f(); }
   void g() { pb->g(); }
   void h() { pa->f(); pb->g(); }
   ....
private:
   A* pa;
   B* pb;
   C* pc;
};
```

This class doesn't look very useful and even seems rather inefficient, since all it is doing is forwarding calls. However, it simplifies code ***elsewhere in the application***, where only the class `D` is visible and calling patterns are simplified. Sometimes, it is possible to put code in `D` to ensure that the other classes are used appropriately.

Design patterns are discussed in more detail later (Lecture 13).

12.7.3 Encapsulate

Early in development, a data entity seemed so simple that the programmer elected to use a `struct`: 530

```
struct Point { double x, double y, double z; };
```

As the software grows, many calculations with points are added. Redundancy appears: for example, two programmers independently discover that they need to compute the distance between two points and add equivalent, but slightly different functions.

The obvious refactoring is to create a class `Point` and to put as many point-related functions into it.

Experienced programmers avoid this problem by not using `struct` in the first place: unless the data is ***really*** simple and is not associated with ***any*** calculations, a `class` is better than a `struct`.

12.7.4 Move Common Code to the Root

Class hierarchies tend to start simple and grow complex. Reviewing a mature class hierarchy often shows that common problems have been solved independently in different derived classes. Find commonalities and move them up the tree, to the root if possible.

12.7.5 Global Names

Global names should be avoided wherever possible, but any complex application is likely to contain a few of them. Global names declared in odd places and imported indiscriminately will cause confusion. Create a class called `GlobalNames` or something similar, and put all global objects into it. Any component that uses globals will then clearly announce that it does so by starting with

```
#include "GlobalNames.h"
```

In modern programming, global names are typically handled through ***dependency injection***.

12.8 Miscellaneous Techniques

There are several techniques for developing well-structured systems or improving systems with structural weaknesses.

12.8.1 CRC Cards

CRC cards where introduced by Kent Beck and Ward Cunningham at OOPSLA (Beck and Cunningham 1989) for teaching programmers the object-oriented paradigm. A CRC card is an index card that is use to represent the responsibilities of classes and the interaction between the classes. The cards are created through scenarios, based on the system requirements, that model the behavior of the system. The name CRC stands for ***Class***, ***Responsibilities***, and ***Collaborators***.[72]

Although CRC cards are old and have largely been replaced by heavy-weight methods such as UML, the underlying ideas are still useful in the early stages of system design. The focus on responsibility and collaboration is appropriate and helps to improve the coupling/cohesion ratio.

12.8.2 DRY

DRY stands for ***don't repeat yourself***. As systems grow, it is easy for them to accumulate multiple implementations of a single function. These functions might vary in small details, such as the number and order of parameters, but they do essentially the same thing. Weed out such repetition by converting all the variations into a single function whenever you can. As (Hunt and Thomas 2000, page 27) say:[73]

> ***Every piece of knowledge must have a unique, unambiguous, authoritative representation within a system.***

12.8.3 YAGNI/KISS

YAGNI stands for ***you aren't going to need it*** and is a slogan associated with agile methods. It is a variant of KISS (Keep It Simple, Stupid). Some programmers are tempted to provide all kinds of features that might come in useful one day. When writing a class, for instance, they will include a lot of methods but they don't actually need right now for the application, but look nice. This is usually a waste of time: there is a good chance that the functions will not in fact be needed and, if they are, it does not take long to write them.

It is tempting to write code that is not necessary right now but (you think) might be needed later. Giving in to this temptation has some disadvantages:[74]

- The time spent is taken from necessary functionality.
- The new features must be debugged, documented, and supported.
- Any new feature imposes constraints on what can be done in the future, so an unnecessary feature now may prevent implementing a necessary feature later.

[72]See http://ootips.org/crc-cards.html
[73]I changed their word "single" to "unique" for greater emphasis.
[74]See http://en.wikipedia.org/wiki/You_ain't_gonna_need_it

- Until the feature is actually needed it is not possible to define what it should do, or to test it. This often results in such features not working right even if they eventually are needed.
- It leads to code bloat; the software becomes larger and more complicated while providing no more functionality.
- Unless there are specifications and some kind of revision control, the feature will never be known to programmers who could make use of it.
- Adding the new feature will inevitably suggest other new features. The result is a snowball effect which can consume unlimited time and resources for no benefit.

References

Beck, K. and W. Cunningham (1989, October). A laboratory for teaching object-oriented thinking. In *Object-Oriented Programming: Systems, Languages and Applications*, pp. 1–6.

Dewhurst, S. C. (2005). *C++ Common Knowledge: Essential Intermediate Programming*. Addison-Wesley.

Fowler, M., K. Beck, J. Brant, W. Opdyke, and D. Roberts (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Object Technology Series. Addison-Wesley.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Hunt, A. and D. Thomas (2000). *The Pragmatic Programmer*. Addison-Wesley.

Kerievsky, J. (2004). *Refactoring to Patterns*. Addison-Wesley.

Lakos, J. (1996). *Large-Scale C++ Software Design*. Professional Computing Series. Addison-Wesley.

Meyers, S. (2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs* (3rd ed.). Addison-Wesley.

Meyers, S. (2014). *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14* (1st ed.). O'Reilly.

Parnas, D. L. (1978). Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd International Conference on Software engineering*, Piscataway, NJ, USA, pp. 264–277. IEEE Press. Reprinted as (Parnas 1979).

Prata, S. (2012). *C++ Primer Plus* (6th ed.). Addison-Wesley.

Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley.

Sutter, H. (2000). *Exceptional C++: 47 Engineering puzzles, Programming Problems, and Solutions*. C++ In-Depth Series. Addison-Wesley.

Sutter, H. (2002). *More Exceptional C++: 40 New Engineering puzzles, Programming Problems, and Solutions*. C++ In-Depth Series. Addison-Wesley.

Weiss, M. A. (2004). *C++ for Java Programmers*. Pearson Prentice Hall.