CHAPTER

# Patterns

# 3

The previous chapter discussed the qualities that differentiate a good API from a bad API. The next couple of chapters focus on the techniques and principles of building high-quality APIs. This particular chapter covers a few useful design patterns and idioms that relate to C++ API design.

A design pattern is a general solution to a common software design problem. The term was made popular by the book *Design Patterns: Elements of Reusable Object-Oriented Software*, also known as the Gang of Four book (Gamma et al., 1994). That book introduced the following list of generic design patterns, organized into three main categories:

| **Creational Patterns** | |
|---|---|
| Abstract Factory | Encapsulates a group of related factories. |
| Builder | Separates a complex object's construction from its representation. |
| Factory Method | Lets a class defer instantiation to subclasses. |
| Prototype | Specifies a prototypical instance of a class that can be cloned to produce new objects. |
| Singleton | Ensures a class has only one instance. |
| **Structural Patterns** | |
| Adapter | Converts the interface of one class into another interface. |
| Bridge | Decouples an abstraction from its implementation so that both can be changed independently. |
| Composite | Composes objects into tree structures to represent part–whole hierarchies. |
| Decorator | Adds additional behavior to an existing object in a dynamic fashion. |
| Façade | Provides a unified higher-level interface to a set of interfaces in a subsystem. |
| Flyweight | Uses sharing to support large numbers of fine-grained objects efficiently. |
| Proxy | Provides a surrogate or placeholder for another object to control access to it. |
| **Behavioral Patterns** | |
| Chain of Responsibility | Gives more than one receiver object a chance to handle a request from a sender object. |
| Command | Encapsulates a request or operation as an object, with support for undoable operations. |
| Interpreter | Specifies how to represent and evaluate sentences in a language. |
| Iterator | Provides a way to access the elements of an aggregate object sequentially. |
| Mediator | Defines an object that encapsulates how a set of objects interact. |

API design for C++.
© 2011 Elsevier Inc. All rights reserved.

| | |
|---|---|
| Memento | Captures an object's internal state so that it can be restored to the same state later. |
| Observer | Allows a one-to-many notification of state changes between objects. |
| State | Allows an object to appear to change its type when its internal state changes. |
| Strategy | Defines a family of algorithms, encapsulates each one, and makes them interchangeable at run time. |
| Template Method | Defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. |
| Visitor | Represents an operation to be performed on the elements of an object structure. |

Since initial publication of the design pattern book in 1994, several more design patterns have been added to this list, including an entire new categorization of concurrency design patterns. The original authors have also recently suggested an improved categorization of core, creational, periphery, and other (Gamma et al., 2009).

However, it is not the intent of this API book to provide coverage of all these design patterns. There are plenty of other books on the market that focus solely on that topic. Instead, I will concentrate on those design patterns that are of particular importance to the design of high-quality APIs and discuss their practical implementation in C++. I will also cover C++ idioms that may not be considered true generic design patterns, but which are nevertheless important techniques for C++ API design. Specifically, I will go into details for the following techniques:

- **Pimpl idiom**. This technique lets you completely hide internal details from your public header files. Essentially, it lets you move private member data and functions to the `.cpp` file. It is therefore an indispensable tool for creating well-insulated APIs.
- **Singleton and Factory Method**. These are two very common creational design patterns that are good to understand deeply. Singleton is useful when you want to enforce that only one instance of an object is ever created. It has some rather tricky implementation aspects in C++ that I will cover, including initialization and multithreading issues. The Factory Method pattern provides a generalized way to create instances of an object and can be a great way to hide implementation details for derived class.
- **Proxy, Adapter, and Façade**. These structural patterns describe various solutions for wrapping an API on top of an existing incompatible or legacy interface. This is often the entire goal of writing an API: to improve the interface of some poorly designed ball of code. Proxy and Adapter patterns provide a one-to-one mapping of new classes to preexisting classes, whereas the Façade provides a simplified interface to a larger collection of classes.
- **Observer**. This behavioral pattern can be used to reduce direct dependencies between classes. It allows conceptually unrelated classes to communicate by allowing one class (the observer) to register for notifications from another class (the subject). As such, this pattern is an important aspect of designing loosely coupled APIs.

In addition to these patterns and idioms, I will also discuss the Visitor behavioral pattern in Chapter 12 at the end of the book. The Visitor pattern gives clients a way to provide their own algorithms to operate on data structures in your API. It is most useful when designing a point of extensibility for your clients, which is why I have deferred it until the extensibility chapter.

## 3.1 PIMPL IDIOM

The term pimpl was first introduced by Jeff Sumner as shorthand for “pointer to implementation” (Sutter, 1999). This technique can be used as a way to avoid exposing private details in your header files (see Figure 3.1). It is therefore an important mechanism to help you maintain a strong separation between your API’s interface and implementation (Sutter and Alexandrescu, 2004). While pimpl is not strictly a design pattern (it’s a workaround to C++ specific limitations), it is an idiom that can be considered a special case of the Bridge design pattern.

If you change one programming habit after reading this book, I hope you’ll choose to pimpl more API code.

**TIP**

Use the pimpl idiom to keep implementation details out of your public header files.

### 3.1.1 Using Pimpl

Pimpl relies on the fact that it’s possible to define a data member of a C++ class that is a pointer to a forward declared type. That is, where the type has only been introduced by name and has not yet been fully defined, thus allowing us to hide the definition of that type within the `.cpp` file. This is often called an opaque pointer because the user cannot see the details for the object being pointed to. In essence, pimpl is a way to employ both logical and physical hiding of your private data members and functions.

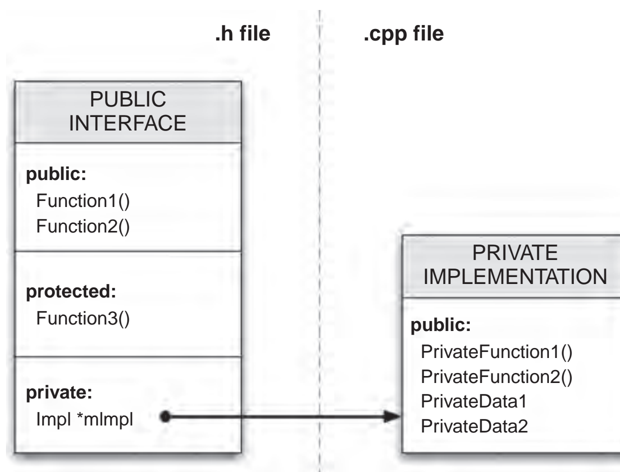


**FIGURE 3.1**

The pimpl idiom, where a public class has a private pointer to a hidden implementation class.

Let's take a look at an example to illustrate this. Consider the following API for an "auto timer": a named object that prints out how long it was alive when it is destroyed.

```
// autotimer.h
#ifdef _WIN32
#include <windows.h>
#else
#include <sys/time.h>
#endif
#include <string>

class AutoTimer
{
public:
   /// Create a new timer object with a human-readable name
   explicit AutoTimer(const std::string &name);
   /// On destruction, the timer reports how long it was alive
   ~AutoTimer();

private:
   // Return how long the object has been alive
   double GetElapsed() const;

   std::string mName;
#ifdef _WIN32
   DWORD mStartTime;
#else
   struct timeval mStartTime;
#endif
};
```

This API violates a number of the important qualities presented in the previous chapter. For example, it includes platform-specific defines and it makes the underlying implementation details of how the timer is stored on different platforms visible to anyone looking at the header file. To be fair, the API does a good job of only exposing the necessary methods as public (i.e., the constructor and destructor) and marking the remaining methods and data members as private. However, C++ requires you to declare these private members in the public header file, which is why you have to include the platform-specific `#if` directives.

What you really want to do is to hide all of the private members in the `.cpp` file. Then you wouldn't need to include any of those bothersome platform specifics. The pimpl idiom lets you do this by placing all of the private members into a class (or struct) that is forward declared in the header but defined in the `.cpp` file. For example, you could recast the aforementioned header as follows using pimpl:

```
// autotimer.h
#include <string>

class AutoTimer
{
```

```
public:
   explicit AutoTimer(const std::string &name);
   ~AutoTimer();

private:
   class Impl;
   Impl *mImpl;
};
```

Now the API is much cleaner! There are no platform-specific preprocessor directives, and the reader cannot see any of the class's private members by looking at the header file.

The implication, however, is that our `AutoTimer` constructor must now allocate an object of type `AutoTimer::Impl` and then destroy it in its destructor. Also, all private members must be accessed via the `mImpl` pointer. However, for most practical cases, the benefit of presenting a clean implementation-free API far outweighs this cost.

To be complete, let's take a look at what the underlying implementation looks like in order to work with this pimpled class. The resulting `.cpp` file looks a little bit messy due to the platform-specific `#ifdef` lines, but the important thing is that this messiness is completely contained in the `.cpp` file now.

```
// autotimer.cpp
#include "autotimer.h"
#include <iostream>
#if _WIN32
#include <windows.h>
#else
#include <sys/time.h>
#endif

class AutoTimer::Impl
{
public:
   double GetElapsed() const
   {
#ifdef _WIN32
   return (GetTickCount() - mStartTime) / 1e3;
#else
   struct timeval end_time;
   gettimeofday(&end_time, NULL);
   double t1 = mStartTime.tv_usec / 1e6 + mStartTime.tv_sec;
   double t2 = end_time.tv_usec / 1e6 + end_time.tv_sec;
   return t2 - t1;
#endif
   }

   std::string mName;
#ifdef _WIN32
```

```
    DWORD mStartTime;
#else
    struct timeval mStartTime;
#endif
};

AutoTimer::AutoTimer(const std::string &name) :
    mImpl(new AutoTimer::Impl())
{
    mImpl->mName = name;
#ifdef _WIN32
    mImpl->mStartTime = GetTickCount();
#else
    gettimeofday(&mImpl->mStartTime, NULL);
#endif
}

AutoTimer::~AutoTimer()
{
    std::cout << mImpl->mName << ": took " << mImpl->GetElapsed()
        << " secs" << std::endl;
    delete mImpl;
    mImpl = NULL;
}
```

Here you see the definition of the `AutoTimer::Impl` class, containing all of the private methods and variables that were originally exposed in the header. Note also that the `AutoTimer` constructor allocates a new `AutoTimer::Impl` object and initializes its members while the destructor deallocates this object.

In the aforementioned design, I declared the `Impl` class as a private nested class within the `AutoTimer` class. Declaring it as a nested class avoids polluting the global namespace with this implementation-specific symbol, and declaring it as private means that it does not pollute the public API of your class. However, declaring it to be private imposes the limitation that only the methods of `AutoTimer` can access members of the `Impl`. Other classes or free functions in the `.cpp` file will not be able to access `Impl`. As an alternative, if this poses too much of a limitation, you could instead declare the `Impl` class to be a public nested class, as in the following example:

```
// autotimer.h
class AutoTimer
{
public:
    explicit AutoTimer(const std::string &name);
    ~AutoTimer();

    // allow access from other classes/functions in autotimer.cpp
    class Impl;
```

```
private:
   Impl *mImpl;
};
```

> **TIP**
>
> When using the pimpl idiom use a private nested implementation class. Only use a public nested Impl class (or a public non-nested class) if other classes or free functions in the .cpp must access Impl members.

Another design question worth considering is how much logic to locate in the Impl class. Some options include:

1. Only private member variables
2. Private member variables and methods
3. All methods of the public class, such that the public methods are simply thin wrappers on top of equivalent methods in the Impl class.

Each of these options may be appropriate under different circumstances. However, in general, I recommend option 2: putting all private member variables and private methods in the `Impl` class. This lets you maintain the encapsulation of data and methods that act on those data and lets you avoid declaring private methods in the public header file. Note that I adopted this design approach in the example given earlier by putting the `GetElapsed()` method inside of the `Impl` class. Herb Sutter notes a couple of caveats with this approach (Sutter, 1999):

1. You can't hide private virtual methods in the implementation class. These must appear in the public class so that any derived classes are able to override them.
2. You may need to add a pointer in the implementation class back to the public class so that the `Impl` class can call public methods. Although you could also pass the public class into the implementation class methods that need it.

### 3.1.2 Copy Semantics

A C++ compiler will create a copy constructor and assignment operator for your class if you don't explicitly define them. However, these default constructors will only perform a shallow copy of your object. This is bad for pimpled classes because it means that if a client copies your object then both objects will point to the same implementation object, `Impl`. However, both objects will attempt to delete this same `Impl` object in their destructors, which will most likely lead to a crash. Two options for dealing with this are as follow.

1. **Make your class uncopyable**. If you don't intend for your users to create copies of an object, then you can declare the object to be non-copyable. You can do this by explicitly declaring a copy constructor and assignment operator. You don't have to provide implementations for these; just the declarations are sufficient to prevent the compiler from generating its own default versions. Declaring these as private is also a good idea so that attempts to copy an object will generate a compile error rather than a link error. Alternatively, if you are using the Boost libraries,

then you could also simply inherit from `boost::noncopyable`. Also, the new C++0x specification lets you disable these default functions completely (see Chapter 6 for details).

2. **Explicitly define the copy semantics**. If you do want your users to be able to copy your pimpled objects, then you should declare and define your own copy constructor and assignment operator. These can then perform a deep copy of your object, that is, create a copy of the Impl object instead of just copying the pointer. I will cover how to write your own constructors and operators in the C++ usage chapter later in this book.

The following code provides an updated version of our `AutoTimer` API where I have made the object be non-copyable by declaring a private copy constructor and assignment operator. The associated `.cpp` file doesn't need to change.

```
#include <string>

class AutoTimer
{
public:
    explicit AutoTimer(const std::string &name);
    ~AutoTimer();

private:
    // Make this object be non-copyable
    AutoTimer(const AutoTimer &);
    const AutoTimer &operator =(const AutoTimer &);

    class Impl;
    Impl *mImpl;
};
```

### 3.1.3 Pimpl and Smart Pointers

One of the inconvenient and error-prone aspects of pimpl is the need to allocate and deallocate the implementation object. Every now and then you may forget to delete the object in your destructor or you may introduce bugs by accessing the `Impl` object before you've allocated it or after you've destroyed it. As a convention, you should therefore ensure that the very first thing your constructor does is to allocate the `Impl` object (preferably via its initialization list), and the very last thing your destructor does is to delete it.

Alternatively, you would rely upon smart pointers to make this a little easier. That is, you could use a shared pointer or a scoped pointer to hold the implementation object pointer. Because a scoped pointer is non-copyable by definition, using this type of smart pointer for objects that you don't want your users to copy would also allow you to avoid having to declare a private copy constructor and assignment operator. In this case, our API can simply appear as:

```
#include <boost/scoped_ptr.hpp>
#include <string>

class AutoTimer
{
```

```
public:
   explicit AutoTimer(const std::string &name);
   ~AutoTimer();

private:
   class Impl;
   boost::scoped_ptr<Impl> mImpl;
};
```

Alternatively, you could use a `boost::shared_ptr`, which would allow the object to be copied without incurring the double delete issues identified earlier. Using a shared pointer would of course mean that any copy would point to the same `Impl` object in memory. If you need the copied object to have a copy of the `Impl` object, then you will still need to write your own copy constructor and assignment operators (or use a copy-on-write pointer, as described in the performance chapter).

> **TIP**
>
> Think about the copy semantics of your pimpl classes and consider using a smart pointer to manage initialization and destruction of the implementation pointer.

Using either a shared or a scoped pointer means that the `Impl` object will be freed automatically when the `AutoTimer` object is destroyed: you no longer need to delete it explicitly in the destructor. So the destructor of our `autotimer.cpp` file can now be reduced to simply:

```
AutoTimer::~AutoTimer()
{
   std::cout << mImpl->mName << ": took " << mImpl->GetElapsed()
             << " secs" << std::endl;
}
```

### 3.1.4 Advantages of Pimpl

There are many advantages to employing the pimpl idiom in your classes. These include the following.

- **Information hiding**. Private members are now completely hidden from your public interface. This allows you to keep your implementation details hidden (and proprietary in the case of closed-source APIs). It also means that your public header files are cleaner and more clearly express the true public interface. As a result, they can be read and digested more easily by your users. One further benefit of information hiding is that your users cannot use dirty tactics as easily to gain access to your private members, such as doing the following, which is actually legal in C++ (Lakos, 1996):

```
#define private public  // make private members be public!
#include "yourapi.h"    // can now access your private members
#undef private          // revert to default private semantics
```

- **Reduced coupling**. As shown in the AutoTimer example earlier, without pimpl, your public header files must include header files for all of your private member variables. In our example, this meant having to include windows.h or sys/time.h. This increases the compile-time coupling of your API on other parts of the system. Using pimpl, you can move those dependencies into the .cpp file and remove those elements of coupling.
- **Faster compiles**. Another implication of moving implementation-specific includes to the .cpp file is that the include hierarchy of your API is reduced. This can have a very direct effect on compile times (Lakos, 1996). I will detail the benefits of minimizing include dependencies in the performance chapter.
- **Greater binary compatibility**. The size of a pimpled object never changes because your object is always the size of a single pointer. Any changes you make to private member variables (recall that member variables should always be private) will only affect the size of the implementation class that is hidden inside of the .cpp file. This makes it possible to make major implementation changes without changing the binary representation of your object.
- **Lazy Allocation**. The mImpl class can be constructed on demand. This may be useful if the class allocates a limited or costly resources such as a network connection.

### 3.1.5 Disadvantages of Pimpl

The primary disadvantage of the pimpl idiom is that you must now allocate and free an additional implementation object for every object that is created. This increases the size of your object by the size of a pointer and may introduce a performance hit for the extra level of pointer indirection required to access all member variables, as well as the cost for additional calls to new and delete. If you are concerned with the memory allocator performance, then you may consider using the "Fast Pimpl" idiom (Sutter, 1999) where you overload the new and delete operators for your Impl class to use a more efficient small-memory fixed-size allocator.

There is also the extra developer inconvenience to prefix all private member accesses with something like mImpl->. This can make the implementation code harder to read and debug due to the additional layer of abstraction. This becomes even more complicated when the Impl class has a pointer back to the public class. You must also remember to define a copy constructor or disable copying of the class. However, these inconveniences are not exposed to users of your API and are therefore not a concern from the point of view of your API's design. They are a burden that you the developer must shoulder in order that all of your users receive a cleaner and more efficient API. To quote a certain science officer and his captain: "The needs of the many outweigh the needs of the few. Or the one."

One final issue to be aware of is that the compiler will no longer catch changes to member variables within const methods. This is because member variables now live in a separate object. Your compiler will only check that you don't change the value of the mImpl pointer in a const method, but not whether you change any members pointed to by mImpl. In effect, every member function of a pimpled class could be defined as const (except of course the constructor or destructor). This is demonstrated by the following const method that legally changes a variable in the Impl object:

```
void PimpledObject::ConstMethod() const
{
    mImpl->mName = "string changed by a const method";
}
```

### 3.1.6 Opaque Pointers in C

While I have focused on C++ so far, you can create opaque pointers in plain C too. The concept is the same: you create a pointer to a struct that is only defined in a `.c` file. The following header file demonstrates what this might look like in C:

```
/* autotimer.h */
/* declare an opaque pointer to an AutoTimer structure */
typedef struct AutoTimer *AutoTimerPtr;

/* functions to create and destroy the AutoTimer structure */
AutoTimerPtr AutoTimerCreate();
void AutoTimerDestroy(AutoTimerPtr ptr);
```

The associated `.c` file may then look as follows:

```
#include "autotimer.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#if _WIN32
#include <windows.h>
#else
#include <sys/time.h>
#endif

struct AutoTimer
{
   char *mName;
#if _WIN32
   DWORD mStartTime;
#else
   struct timeval mStartTime;
#endif
} AutoTimer;

AutoTimerPtr AutoTimerCreate(const char *name)
{
   AutoTimerPtr ptr = malloc(sizeof(AutoTimer));
   if (ptr)
   {
      ptr->mName = strdup(name);
#if _WIN32
   ptr->mStartTime = GetTickCount();
#else
   gettimeofday(&ptr->mStartTime, NULL);
#endif
   }
   return ptr;
}
```

```
static double GetElapsed(AutoTimerPtr ptr)
{
#if _WIN32
   return (GetTickCount() - ptr->mStartTime) / 1e3;
#else
   struct timeval end_time;
   gettimeofday(&end_time, NULL);
   double t1 = ptr->mStartTime.tv_usec / 1e6 +
          ptr->mStartTime.tv_sec;
   double t2 = end_time.tv_usec / 1e6 + end_time.tv_sec;
   return t2 - t1;
#endif
}

void AutoTimerDestroy(AutoTimerPtr ptr)
{
   if (ptr)
   {
      printf("%s: took %f secs\n", ptr->mName, GetElapsed(ptr));
      free(ptr);
   }
}
```

## 3.2 SINGLETON

The Singleton design pattern (Gamma et al., 1994) is used to ensure that a class only ever has one instance. The pattern also provides a global point of access to that single instance (Figure 3.2). You can think of a singleton as a more elegant global variable. However, it offers several advantages over the use of global variables because it

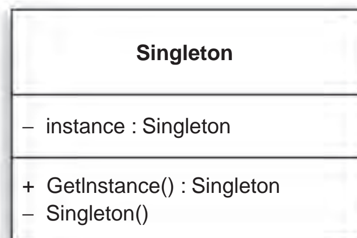


**FIGURE 3.2**

UML diagram of the Singleton design pattern.

1. Enforces that only one instance of the class can be created.
2. Provides control over the allocation and destruction of the object.
3. Allows support for thread-safe access to the object's global state.
4. Avoids polluting the global namespace.

The Singleton pattern is useful for modeling resources that are inherently singular in nature. For example, a class to access the system clock, the global clipboard, or the keyboard. It's also useful for creating manager classes that provide a single point of access to multiple resources, such as a thread manager or an event manager. However, the singleton is still essentially a way to add global variables to your system, albeit in a more manageable fashion. It can therefore introduce global state and dependencies into your API that are difficult to refactor later, as well as making it difficult to write unit tests that exercise isolated parts of your code.

I have decided to cover the concept of singletons here partly because they offer a useful and common API design technique. However, another reason is that because they are fairly intricate to implement robustly in C++, it's worth discussing some of the implementation details. Also, because many programmers have a tendency to overuse the Singleton pattern, I wanted to highlight some of the disadvantages of singletons as well as provide alternative techniques.

> **TIP**
>
> A Singleton is a more elegant way to maintain global state, but you should always question whether you need global state.

### 3.2.1 Implementing Singletons in C++

The Singleton pattern involves creating a class with a static method that returns the same instance of the class every time it is called. This static method is often called `GetInstance()`, or similar. There are several C++ language features to consider when designing a singleton class.

- You don't want clients to be able to create new instances. This can be done by declaring the default constructor to be private, thus preventing the compiler from automatically creating it as public.
- You want the singleton to be non-copyable, to enforce that a second instance cannot be created. As seen earlier, this can be done by declaring a private copy constructor and a private assignment operator.
- You want to prevent clients from being able to delete the singleton instance. This can be done by declaring the destructor to be private. (Note, however, that some compilers, such as Borland 5.5 and Visual Studio 6, produce an error incorrectly if you try to declare a destructor as private.)
- The `GetInstance()` method could return either a pointer or a reference to the singleton class. However, if you return a pointer, clients could potentially delete the object. You should therefore prefer returning a reference.

The general form of a singleton in C++ can therefore be given as follows (Alexandrescu, 2001):

```
class Singleton
{
public:
   static Singleton &GetInstance();
```

```
private:
   Singleton();
   ~Singleton();
   Singleton(const Singleton &);
   const Singleton &operator =(const Singleton &);
};
```

Then user code can request a reference to the singleton instance as follows:

```
Singleton &obj = Singleton::GetInstance();
```

Note that declaring the constructor and destructor to be private also means that clients cannot create subclasses of the singleton. However, if you wish to allow this, you can of course simply declare them to be protected instead.

> **TIP**
>
> Declare the constructor, destructor, copy constructor, and assignment operator to be private (or protected) to enforce the Singleton property.

In terms of implementation, one of the areas to be very careful about is how the singleton instance is allocated. The important C++ initialization issue to be cognizant of is explained by Scott Meyers as follows:

> *The relative order of initialization of non-local static objects in different translation units is undefined (Meyers, 2005).*

This means that it would be dangerous to initialize our singleton using a non-local static variable. A non-local object is one that is declared outside of a function. Static objects include global objects and objects declared as static inside of a class, function, or a file scope. As a result, one way to initialize our singleton would be to create a static variable inside a method of our class, as follows:

```
Singleton &Singleton::GetInstance()
{
   static Singleton instance;
   return instance;
}
```

One nice property of this approach is that the instance will only be allocated when the `GetInstance()` method is first called. This means that if the singleton is never requested, the object is never allocated. However, on the down side, this approach is not thread safe. Also, Andrei Alexandrescu notes that this technique relies on the standard last-in-first-out deallocation behavior of static variables, which can result in singletons being deallocated before they should in situations where singletons call other singletons in their destructors. As an example of this problem, consider two singletons: `Clipboard` and `LogFile`. When `Clipboard` is instantiated, it also instantiates `LogFile` to output some diagnostic information. At program exit, `LogFile` is destroyed first because

it was created last and then `Clipboard` is destroyed. However, the `Clipboard` destructor tries to call `LogFile` to log the fact that it is being destroyed, but `LogFile` has already been freed. This will most likely result in a crash on program exit.

In his *Modern C++ Design* book, Alexandrescu presents several solutions to this destruction order problem, including resurrecting the singleton if it is needed after it has been destroyed, increasing the longevity of a singleton so that it can outlive other singletons, and simply not deallocating the singleton (i.e., relying on the operating system to free all allocated memory and close any file handles). If you find yourself needing to implement one of these solutions, I refer you to this book for details (Alexandrescu, 2001).

### 3.2.2 Making Singletons Thread Safe

The implementation of `GetInstance()` presented earlier is not thread safe because there is a race condition in the initialization of the `Singleton` static. If two threads happen to call this method at the same time, then the instance could be constructed twice or it could be used by one thread before it has been fully initialized by the other thread. This race condition is more evident if you look at the code that the compiler will generate for this method. Here's an example of what the `GetInstance()` method might get expanded to by a compiler:

```
Singleton &Singleton::GetInstance()
{
   // Example code that a compiler might generate...
   extern void __DestructSingleton();
   static char __buffer[sizeof(Singleton)];
   static bool __initialized = false;
   if (! __initialized)
   {
      new(__buffer) Singleton();    // placement new syntax
      atexit(__DestructSingleton); // destroy instance on exit
      __initialized = true;
   }
   return *reinterpret_cast<Singleton *>(__buffer);
}

void __DestructSingleton()
{
   // call destructor for static __buffer Singleton object
}
```

As with most solutions to non-thread-safe code, you can make this method thread safe by adding a mutex lock around the code that exhibits the race condition:

```
Singleton &Singleton::GetInstance()
{
   Mutex mutex;
   ScopedLock(&mutex);   // unlocks mutex on function exit
```

```
    static Singleton instance;
    return instance;
}
```

The potential problem with this solution is that it may be expensive because the lock will be acquired every time the method is called. It should be noted that this may not actually be a performance issue for your API. Always measure performance in real-world uses before deciding to optimize. For example, if this method is not called frequently by your clients, then this solution should be perfectly acceptable. As a workaround for clients who report performance problems, you could suggest that they call this method once (or once per thread) and cache the result in their own code. However, if the performance of this method really is an issue for you, then you're going to have to get a bit more complicated.

A commonly proposed solution to optimize this kind of over aggressive locking behavior is to use the Double Check Locking Pattern (DCLP), which looks like:

```
Singleton &Singleton::GetInstance()
{
    static Singleton *instance = NULL;

    if (! instance)    // check #1
    {
        Mutex mutex;
        ScopedLock(&mutex);

        if (! instance) // check #2
        {
            instance = new Singleton();
        }
    }

    return *instance;
}
```

However, the DCLP is not guaranteed to work on all compilers and under all processor memory models. For example, a shared-memory symmetric multiprocessor normally commits writes to memory in bursts, which may cause the writes for different threads to be reordered. Using the `volatile` keyword is often seen as a solution to this problem because it synchronizes read and write operations to the volatile data. However, even this approach can be flawed in a multithreaded environment (Meyers and Alexandrescu, 2004). You may be able to use platform-specific memory barriers to get around these problems or, if you're only using POSIX threads, you could use `pthread_once()`, but at this point it's probably worth stepping back a bit and recognizing that perhaps you simply shouldn't try to optimize the `GetInstance()` method as formulated earlier. The various compiler and platform idiosyncrasies mean that your API may work fine for some clients, but will fail in complex and difficult-to-debug ways for other clients. Ultimately these difficulties are a product of trying to enforce thread safety in a language that has no inherent awareness or support for concurrency.

If the performance of a thread-safe `GetInstance()` is critical to you, then you might consider avoiding the lazy instantiation model presented earlier and instead initialize your singleton on startup, for example, either before `main()` is called or via a mutex-locked API initialization call.

One benefit common to both of these options is that you don't have to change the implementation of your `Singleton` class to support multithreading.

1. **Static initialization**. Static initializers get called before `main()`, where you can normally assume that the program is still single threaded. As a result, you could create your singleton instance as part of a static initializer and avoid the need for any mutex locking. Of course, you need to make sure that your constructor doesn't depend on non-local static variables in other `.cpp` files. However, bearing this caveat in mind, you could add the following static initialization call to the `singleton.cpp` file to ensure that the instance is created before `main()` is called.

   ```
   static Singleton &foo = Singleton::GetInstance();
   ```

2. **Explicit API initialization**. You could consider adding an initialization routine for your library if you don't already have one. In this case, you could remove the mutex locks from the `GetInstance()` method and instead instantiate the singleton as part of this library initialization routine, placing the mutex locks at this point.

   ```
   void APIInitialize()
   {
      Mutex mutex;
      ScopedLock(&mutex);

      Singleton::GetInstance();
   }
   ```

   This has the benefit that you can specify the order of initialization of all your singletons, in case you have a singleton dependency issue (hopefully you don't). While it is somewhat ugly to require users to explicitly initialize your library, recall that this is only necessary if you need to provide a thread-safe API.

> **TIP**
>
> Creating a thread-safe Singleton in C++ is hard. Consider initializing it with a static constructor or an API initialization function.

### 3.2.3 Singleton versus Dependency Injection

Dependency injection is a technique where an object is passed into a class (injected) instead of having the class create and store the object itself. Martin Fowler coined the term in 2004 as a more specific form of the Inversion of Control concept. As a simple example, consider the following class that depends on a database object:

```
class MyClass
{
   MyClass() :
      mDatabase(new Database("mydb", "localhost", "user", "pass"))
   {}
```

```
private:
   Database *mDatabase;
};
```

The problem with this approach is that if the `Database` constructor is changed or if someone changes the password for the account called "user" in the live database, then you will have to change `MyClass` to fix the problem. Also, from an efficiency point of view, every instance of `MyClass` will create a new `Database` instance. As an alternative, you can use dependency injection to pass a preconfigured `Database` object into `MyClass`, as follows:

```
class MyClass
{
   MyClass(Database *db) :
      mDatabase(db)
   {}

private:
   Database *mDatabase;
};
```

In this way, `MyClass` no longer has to know how to create a `Database` instance. Instead, it gets passed an already constructed and configured `Database` object for it to use. This example demonstrates constructor injection, that is, passing the dependent object via the constructor, but you could just as easily pass in dependencies via a setter member function or you could even define a reusable interface to inject certain types of objects and then inherit from that interface in your classes.

Of course, the `Database` object needs to be created somewhere. This is typically the job of a dependency container. For example, a dependency container would be responsible for creating instances of the `MyClass` class and passing it an appropriate `Database` instance. In other words, a dependency container can be thought of as a generic factory class. The only real difference between the two is that a dependency container will maintain state, such as the single `Database` instance in our case.

**TIP**

Dependency injection makes it easier to test code that uses Singletons.

Dependency injection can therefore be viewed as a way to avoid the proliferation of singletons by encouraging interfaces that accept the single instance as an input rather than requesting it internally via a `GetInstance()` method. This also makes for more testable interfaces because the dependencies of an object can be substituted with stub or mock versions for the purposes of unit testing (this is discussed further in the testing chapter).

### 3.2.4 Singleton versus Monostate

Most problems associated with the Singleton pattern derive from the fact that it is designed to hold and control access to global state. However, if you don't need to control when the state is initialized or

don't need to store state in the singleton object itself, then other techniques can be used, such as the Monostate design pattern.

The Monostate pattern allows multiple instances of a class to be created where all of those instances use the same static data. For instance, here's a simple case of the Monostate pattern:

```
// monostate.h
class Monostate
{
public:
   int GetTheAnswer() const { return sAnswer; }

private:
   static int sAnswer;
};

// monostate.cpp
int Monostate::sAnswer = 42;
```

In this example, you can create multiple instances of the `Monostate` class, but all calls to the `GetTheAnswer()` method will return the same result because all instances share the same static variable `sAnswer`. You could also hide the declaration of the static variable from the header completely by just declaring it as a file-scope static variable in `monostate.cpp` instead of a private class static variable. Because static members do not contribute to the per instance size of a class, doing this will have no physical impact on the API, other than to hide implementation details from the header.

Some benefits of the Monostate pattern are that it

- Allows multiple instances to be created.
- Offers transparent usage because no special `GetInstance()` method is needed.
- Exhibits well-defined creation and destruction semantics using static variables.

As Robert C. Martin notes, Singleton enforces the structure of singularity by only allowing one instance to be created, whereas Monostate enforces the behavior of singularity by sharing the same data for all instances (Martin, 2002).

**TIP**

Consider using Monostate instead of Singleton if you don't need lazy initialization of global data or if you want the singular nature of the class to be transparent.

As a further real-world example, the Second Life source code uses the Monostate pattern for its `LLWeb` class. This example uses a version of Monostate where all member functions are declared static.

```
class LLWeb
{
public:
   static void InitClass();

   /// Load the given url in the user's preferred web browser
```

```
    static void LoadURL(const std::string& url);

    /// Load the given url in the Second Life internal web browser
    static void LoadURLInternal(const std::string &url);

    /// Load the given url in the operating system's web browser
    static void LoadURLExternal(const std::string& url);

    /// Returns escaped url (eg, " " to "%20")
    static std::string EscapeURL(const std::string& url);
};
```

In this case, `LLWeb` is simply a manager class that provides a single access point to the functionality for opening Web pages. The actual Web browser functionality itself is implemented in other classes. The `LLWeb` class does not hold any state itself, although of course internally any of the static methods could access static variables.

One of the drawbacks with this static method version of Monostate is that you cannot subclass any of the static methods because static member functions cannot be virtual. Also, because you no longer instantiate the class, you cannot write a constructor or destructor to perform any initialization or cleanup. This is necessary in this case because `LLWeb` accesses dynamically allocated global state instead of relying on static variables that are initialized by the compiler. The creator of `LLWeb` got around this limitation by introducing an `initClass()` static method that requires a client program to initialize the class explicitly. A better design may have been to hide this call within the `.cpp` file and invoke it lazily from each of the public static methods. However, in that case, the same thread safety concerns raised earlier would be applicable.

### 3.2.5 Singleton versus Session State

In a recent retrospective interview, authors of the original design patterns book stated that the only pattern they would consider removing from the original list is Singleton. This is because it is essentially a way to store global data and tends to be an indicator of poor design (Gamma et al., 2009).

Therefore, as a final note on the topic of singletons, I urge you to really think about whether a singleton is the correct pattern for your needs. It's often easy to think that you will only ever need a single instance of a given class. However, requirements change and code evolves, and in the future you may find that you need to support multiple instances of the class.

For example, consider that you are writing a simple text editor. You use a singleton to hold the current text style (e.g., bold, italics, underlined) because the user can only ever have one style active at one time. However, this restriction is only valid because of the initial assumption that the program can edit only one document at a time. In a later version of the program, you are asked to add support for multiple documents, each with their own current text style. Now you have to refactor your code to remove the singleton. Ultimately, singletons should only be used to model objects that are truly singular in their nature. For example, because there is only one system clipboard, it may still be reasonable to model the clipboard for the text editor as a singleton.

Often it's useful to think about introducing a "session" or "execution context" object into your system early on. This is a single instance that holds all of the state for your code rather than representing that state with multiple singletons. For example, in the text editor example, you might

introduce a `Document` object. This will have accessors for things such as the current text style, but those objects do not have to be enforced as singletons. They are just plain classes that can be accessed from the `Document` class as `document->GetTextStyle()`. You can start off with a single `Document` instance, accessible by a call such as `Document::GetCurrent()` for instance. You might even implement `Document` as a singleton to begin with. However, if you later need to add support for multiple contexts (i.e., multiple documents), then your code is in a much healthier state to support this because you only have one singleton to refactor instead of dozens. J.B. Rainsberger refers to this as a Toolbox Singleton, where the application becomes the singleton, not the individual classes (Rainsberger, 2001).

**TIP**

There are several alternatives to the Singleton pattern, including dependency injection, the Monostate pattern, and use of a session context.

## 3.3 FACTORY METHODS

A Factory Method is a creational design pattern that allows you to create objects without having to specify the specific C++ type of the object to create. In essence, a factory method is a generalization of a constructor. Constructors in C++ have several limitations, such as the following.

1. **No return result**. You cannot return a result from a constructor. This means that you cannot signal an error during the initialization of an object by returning a NULL pointer, for instance (although you can signal an error by throwing an exception within a constructor).
2. **Constrained naming**. A constructor is easily distinguished because it is constrained to have the same name as the class that it lives in. However, this also limits its flexibility. For example, you cannot have two constructors that both take a single integer argument.
3. **Statically bound creation**. When constructing an object, you must specify the name of a concrete class that is known at compile time, for example, you might write: `Foo *f = new Foo()`, where `Foo` is a specific type that must be known by the compiler. There is no concept of dynamic binding at run time for constructors in C++.
4. **No virtual constructors**. You cannot declare a virtual constructor in C++. As just noted, you must specify the exact type of the object to be constructed at compile time. The compiler therefore allocates the memory for that specific type and then calls the default constructor for any base classes (unless you explicitly specify a non-default constructor in the initialization list). It then calls the constructor for the specific type itself. This is also why you cannot call virtual methods from the constructor and expect them to call the derived override (because the derived class hasn't been initialized yet).

In contrast, factory methods circumvent all of these limitations. At a basic level, a factory method is simply a normal method call that can return an instance of a class. However, they are often used in combination with inheritance, where a derived class can override the factory method and return an instance of that derived class. It's also very common and useful to implement factories using abstract base classes (ABC) (DeLoura, 2001). I will focus on the abstract base class

case here, so let's recap what these kinds of classes are before I dive further into using factory methods.

### 3.3.1 Abstract Base Classes

An ABC is a class that contains one or more pure virtual member functions. Such a class is not concrete and cannot be instantiated using the new operator. Instead, it is used as a base class where derived classes provide the implementations of the pure virtual methods. For example,

```
// renderer.h
#include <string>

class IRenderer
{
public:
    virtual ~IRenderer() {}
    virtual bool LoadScene(const std::string &filename) = 0;
    virtual void SetViewportSize(int w, int h) = 0;
    virtual void SetCameraPosition(double x, double y, double z) = 0;
    virtual void SetLookAt(double x, double y, double z) = 0;
    virtual void Render() = 0;
};
```

This defines an abstract base class to describe an extremely simple 3D graphics renderer. The "= 0" suffix on the methods declares them as pure virtual methods, meaning that they must be overridden in a derived class for that class to be concrete. Note that it is not strictly true to say that pure virtual methods provide no implementation. You can actually provide a default implementation for pure virtual methods in your `.cpp` file. For example, you could provide an implementation for `SetViewportSize()` in `renderer.cpp` and then a derived class would be able to call `IRenderer::SetViewportSize()`, although it would still have to explicitly override the method as well.

An abstract base class is therefore useful to describe abstract units of behaviors that can be shared by multiple classes; it specifies a contract that all concrete derived classes must conform to. In Java, this is referred to as an interface (with the constraint that Java interfaces can only have public methods, static variables, and they cannot define constructors). I have named the aforementioned `IRenderer` class with an "I" prefix to indicate that it's an interface class.

Of course, you can also provide methods with implementations in the abstract base class: not all of the methods have to be pure virtual. In this regard, abstract base classes can be used to simulate mixins, which can be thought of loosely as interfaces with implemented methods.

As with any class that has one or more virtual methods, you should always declare the destructor of an abstract base class to be virtual. The following code illustrates why this is important.

```
class IRenderer
{
    // no virtual destructor declared
    virtual void Render() = 0;
};
```

```
class RayTracer : public IRenderer
{
   RayTracer();
   ~RayTracer();
   void Render(); // provide implementation for ABC method
};

int main(int, char **)
{
   IRenderer *r = new RayTracer();
   // delete calls IRenderer::~IRenderer, not RayTracer::~RayTracer
   delete r;
}
```

### 3.3.2 Simple Factory Example

Now that I have reviewed what an abstract base class is, let's use it to provide a simple factory method. I'll continue with the `renderer.h` example given earlier and start by declaring the factory for objects of type `IRenderer`.

```
// rendererfactory.h
#include "renderer.h"
#include <string>

class RendererFactory
{
public:
   IRenderer *CreateRenderer(const std::string &type);
};
```

That's all there is to declaring a factory method: it's just a normal method that can return an instance of an object. Note that this method cannot return an instance of the specific type `IRenderer` because that's an abstract base class and cannot be instantiated. However, it can return instances of derived classes. Also, you can use the string argument to `CreateRenderer()` to specify which derived type you want to create.

Let's assume that you have implemented three concrete classes derived from `IRenderer`: `OpenGLRenderer`, `DirectXRenderer`, and `MesaRenderer`. Let's further specify that you don't want users of your API to have any knowledge of the existence of these types: they must be completely hidden behind the API. Based on these conditions, you can provide an implementation of the factory method as follows:

```
// rendererfactory.cpp
#include "rendererfactory.h"
#include "openglrenderer.h"
#include "directxrenderer.h"
```

```
#include "mesarenderer.h"

IRenderer *RendererFactory::CreateRenderer(const std::string &type)
{
    if (type == "opengl")
        return new OpenGLRenderer();

    if (type == "directx")
        return new DirectXRenderer();

    if (type == "mesa")
        return new MesaRenderer();

    return NULL;
}
```

This factory method can therefore return any of the three derived classes of `IRenderer`, depending on the type string that the client passes in. This lets users decide which derived class to create at run time, not compile time as a normal constructor requires you to do. This is an enormous advantage because it means that you can create different classes based on user input or on the contents of a configuration file that is read at run time.

Also, note that the header files for the various concrete derived classes are only included in the factory's `.cpp` file. They do not appear in the `rendererfactory.h` public header. In effect, these are private header files and do not need to be distributed with your API. As such, users can never see the private details of your different renderers, and they can't even see the specific types used to implement these different renderers. Users only ever specify a renderer via a string variable (or an enum, if you prefer).

**TIP**

Use Factory Methods to provide more powerful class construction semantics and to hide subclass details.

This example demonstrates a perfectly acceptable factory method. However, one potential drawback is that it contains hardcoded knowledge of the available derived classes. If you add a new renderer to the system, you have to edit `rendererfactory.cpp`. This is not terribly burdensome, and most importantly it will not affect our public API. However, it does mean that you cannot add support for new derived classes at run time. More specifically, it means that your users cannot add new renderers to the system. These concerns are addressed by presenting an extensible object factory.

### 3.3.3 Extensible Factory Example

To decouple the concrete derived classes from the factory method and to allow new derived classes to be added at run time, you can update the factory class to maintain a map that associates type names to object creation callbacks (Alexandrescu, 2001). You can then allow new derived classes to be registered and unregistered using a couple of new method calls. The ability to register new

classes at run time allows this form of the Factory Method pattern to be used to create extensible plugin interfaces for your API, as detailed in Chapter 12.

One further issue to note is that the factory object must now hold state. As such, it would be best to enforce that only one factory object is ever created. This is the reason why most factory objects are also singletons. In the interests of simplicity, however, I will use static methods and variables in our example here. Putting all of these points together, here's what our new object factory might look like:

```
// rendererfactory.h
#include "renderer.h"
#include <string>
#include <map>

class RendererFactory
{
public:
   typedef IRenderer *(*CreateCallback)();
   static void RegisterRenderer(const std::string &type, CreateCallback cb);
   static void UnregisterRenderer(const std::string &type);
   static IRenderer *CreateRenderer(const std::string &type);

private:
   typedef std::map<std::string, CreateCallback> CallbackMap;
   static CallbackMap mRenderers;
};
```

For completeness, the associated `.cpp` file might look like:

```
#include "rendererfactory.h"

// instantiate the static variable in RendererFactory
RendererFactory::CallbackMap RendererFactory::mRenderers;

void RendererFactory::RegisterRenderer(const std::string &type, CreateCallback cb)
{
   mRenderers[type] = cb;
}

void RendererFactory::UnregisterRenderer(const std::string &type)
{
   mRenderers.erase(type);
}

IRenderer *RendererFactory::CreateRenderer(const std::string &type)
```

```
{
    CallbackMap::iterator it = mRenderers.find(type);
    if (it != mRenderers.end())
    {
        // call the creation callback to construct this derived type
        return (it->second)();
    }
    return NULL;
}
```

A user of your API can now register (and unregister) new renderers in your system. The compiler will ensure that the user's new renderer conforms to your `IRenderer` abstract interface, that is, it provides an implementation for all of the pure virtual methods in `IRenderer`. To illustrate this, the following code shows how a user could define their own renderer, register it with the object factory, and then ask the factory to create an instance of it.

```
class UserRenderer : public IRenderer
{
public:
    ~UserRenderer() {}
    bool LoadScene(const std::string &filename) { return true; }
    void SetViewportSize(int w, int h) {}
    void SetCameraPosition(double x, double y, double z) {}
    void SetLookAt(double x, double y, double z) {}
    void Render() { std::cout << "User Render" << std::endl; }
    static IRenderer *Create() { return new UserRenderer(); }
};

int main(int, char **)
{
    // register a new renderer
    RendererFactory::RegisterRenderer("user", UserRenderer::Create);

    // create an instance of our new renderer
    IRenderer *r = RendererFactory::CreateRenderer("user");
    r->Render();
    delete r;

    return 0;
}
```

One point worth noting here is that I added a `Create()` function to the `UserRenderer` class. This is because the register method of the factory needs to take a callback that returns an object. This callback doesn't have to be part of the `IRenderer` class (it could be a free function, for example). However, adding it to the `IRenderer` class is a good idea to keep all of the related functionality in the same place. In fact, you could even enforce this convention by adding the `Create()` call as another pure virtual method on the `IRenderer` abstract base class.

Finally, I note that in the extensible factory example given here, a renderer callback has to be visible to the `RegisterRenderer()` function at run time. However, this doesn't mean that you have to expose the built-in renderers of your API. These can still be hidden either by registering them within your API initialization routine or by using a hybrid of the simple factory and the extensible factory, whereby the factory method first checks the type string against a few built-in names. If none of those match, it then checks for any names that have been registered by the user. This hybrid approach has the potentially desirable behavior that users cannot override your built-in classes.

## 3.4 API WRAPPING PATTERNS

Writing a wrapper interface that sits on top of another set of classes is a relatively common API design task. For example, perhaps you are working with a large legacy code base and rather than rearchitecting all of that code you decide to design a new cleaner API that hides the underlying legacy code (Feathers, 2004). Or perhaps you have written a C++ API and need to expose a plain C interface for certain clients. Or perhaps you have a third-party library dependency that you want your clients to be able to access but you don't want to expose that library directly to them.

The downside of creating a wrapper API is the potential performance hit that you may experienced due to the extra level of indirection and the overhead of any extra state that needs to be stored at the wrapper level. However, this is often worth the cost in order to expose a higher-quality or more focused API, such as in the cases just mentioned.

Several structural design patterns deal with the task of wrapping one interface on top of another. I will describe three of these patterns in the following sections. These are, in increasing deviation between the wrapper layer and the original interface: Proxy, Adapter, and Façade.

### 3.4.1 The Proxy Pattern

The Proxy design pattern (Figure 3.3) provides a one-to-one forwarding interface to another class: calling `FunctionA()` in the proxy class will cause it to call `FunctionA()` in the original class. That is, the proxy class and the original class have the same interface. This can be thought of as a single-component wrapper, to use the terminology of Lakos (1996), that is, a single class in the proxy API maps to a single class in the original API.

This pattern is often implemented by making the proxy class store a copy of, or more likely a pointer to, the original class. Then the methods of the proxy class simply redirect to the method with

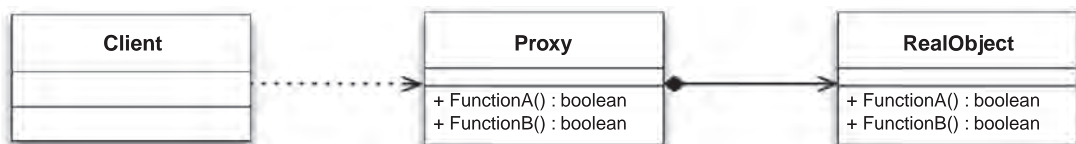


**FIGURE 3.3**

UML diagram of the Proxy design pattern.

the same name in the original object. A downside of this technique is the need to reexpose functions in the original object, a process that essentially equates to code duplication. This approach therefore requires diligence to maintain the integrity of the proxy interface when making changes to the original object. The following code provides a simple example of this technique. Note that I declare the copy constructor and assignment operator as private member functions to prevent clients from copying the object. You could of course allow copying by providing explicit implementations of these functions. I will cover how to do this in the later chapter on C++ usage.

```
class Proxy
{
public:
    Proxy() : mOrig(new Original())
    {}
    ~Proxy()
    {
        delete mOrig;
    }

    bool DoSomething(int value)
    {
        return mOrig->DoSomething(value);
    }

private:
    Proxy(const Proxy &);
    const Proxy &operator =(const Proxy &);

    Original *mOrig;
};
```

An alternative solution is to augment this approach by using an abstract interface that is shared by both the proxy and original APIs. This is done to try and better keep the two APIs synchronized, although it requires you to be able to modify the original API. The following code demonstrates this approach:

```
class IOriginal
{
public:
    virtual bool DoSomething(int value) = 0;
};

class Original : public IOriginal
{
public:
    bool DoSomething(int value);
};

class Proxy : public IOriginal
```

```
{
public:
   Proxy() : mOrig(new Original())
   {}
   ~Proxy()
   {
      delete mOrig;
   }

   bool DoSomething(int value)
   {
      return mOrig->DoSomething(value);
   }

private:
   Proxy(const Proxy &);
   const Proxy &operator =(const Proxy &);

   Original *mOrig;
};
```

**TIP**

A Proxy provides an interface that forwards function calls to another interface of the same form.

A Proxy pattern is useful to modify the behavior of the `Original` class while still preserving its interface. This is particularly useful if the `Original` class is in a third-party library and hence not easily modifiable directly. Some use cases for the proxy pattern include the following.

1. **Implement lazy instantiation of the Original object**. In this case, the `Original` object is not actually instantiated until a method call is invoked. This can be useful if instantiating the `Original` object is a heavyweight operation that you wish to defer until absolutely necessary.
2. **Implement access control to the Original object**. For example, you may wish to insert a permissions layer between the `Proxy` and the `Original` objects to ensure that users can only call certain methods on the `Original` object if they have the appropriate permission.
3. **Support debug or "dry run" modes**. This lets you insert debugging statements into the `Proxy` methods to log all calls to the `Original` object or you can stop the forwarding to certain `Original` methods with a flag to let you call the `Proxy` in a dry run mode; for example, to turn off writing the object's state to disk.
4. **Make the `Original` class be thread safe**. This can be done by adding mutex locking to the relevant methods that are not thread safe. While this may not be the most efficient way to make the underlying class thread safe, it is a useful stop gap if you cannot modify `Original`.
5. **Support resource sharing**. You could have multiple `Proxy` objects share the same underlying `Original` class. For example, this could be used to implement reference counting or copy-on-write semantics. This case is actually another design pattern, called the Flyweight pattern, where multiple objects share the same underlying data to minimize memory footprint.

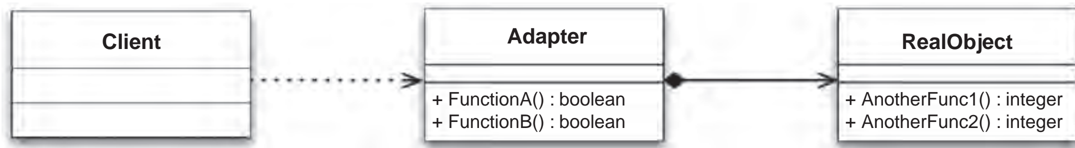


**FIGURE 3.4**

UML diagram of the Adapter design pattern.

6. **Protect against future changes in the `Original` class**. In this case, you anticipate that a dependent library will change in the future so you create a proxy wrapper around that API that directly mimics the current behavior. When the library changes in the future, you can preserve the old interface via your proxy object and simply change its underlying implementation to use the new library methods. At which point, you will no longer have a proxy object, but an adapter, which is a nice segue to our next pattern.

### 3.4.2 The Adapter Pattern

The Adapter design pattern (Figure 3.4) translates the interface for one class into a compatible but different interface. This is therefore similar to the Proxy pattern in that it's a single-component wrapper. However, the interface for the adapter class and the original class may be different.

This pattern is useful to be able to expose a different interface for an existing API to allow it to work with other code. As in the case for the proxy pattern, the two interfaces in question could come from different libraries. For example, consider a geometry package that lets you define a series of primitive shapes. The parameters for certain methods may be in a different order from those that you use in your API, or they may be specified in a different coordinate system, or using a different convention such as (center, size) versus (bottom-left, top-right), or the method names may not follow your API's naming convention. You could therefore use an adapter class to convert this interface into a compatible form for your API. For example,

```
class RectangleAdapter
{
public:
   RectangleAdapter() :
      mRect(new Rectangle())
   {}
   ~RectangleAdapter()
   {
      delete mRect;
   }

   void Set(float x1, float y1, float x2, float y2)
   {
      float w = x2 - x1;
      float h = y2 - y1;
```

```
        float cx = w / 2.0f + x1;
        float cy = h / 2.0f + y1;
        mRect->setDimensions(cx, cy, w, h);
    }

private:
    RectangleAdapter(const RectangleAdapter &);
    const RectangleAdapter &operator =(const RectangleAdapter &);

    Rectangle *mRect;
};
```

In this example, the `RectangleAdapter` uses a different method name and calling conventions to set the dimensions of the rectangle than the underlying `Rectangle` class. The functionality is the same in both cases. You're just exposing a different interface to allow you to work with the class more easily.

**TIP**

An Adapter translates one interface into a compatible but different interface.

It should be noted that adapters can be implemented using composition (as in the aforementioned example) or inheritance. These two flavors are often referred to as object adapters or class adapters, respectively. In the inheritance case, `RectangleAdapter` would derive from the `Rectangle` base class. This could be done using public inheritance if you wanted to also expose the interface of `Rectangle` in your adapter API, although it is more likely that you would use private inheritance so that only your new interface is made public.

Some benefits of the adapter pattern for API design include the following.

1. **Enforce consistency across your API**. As discussed in the previous chapter, consistency is an important quality of good APIs. Using an adapter pattern, you can collate multiple disparate classes that all have different interface styles and provide a consistent interface to all of these. The result is that your API is more uniform and therefore easier to use.
2. **Wrap a dependent library of your API**. For example, your API may provide the ability to load a PNG image. You want to use the libpng library to implement this functionality, but you don't want to expose the libpng calls directly to the users of your API. This could be because you want to present a consistent and uniform API or because you want to protect against potential future API changes in libpng.
3. **Transform data types**. For example, consider that you have an API, `MapPlot`, that lets you plot geographic coordinates on a 2D map. `MapPlot` only accepts latitude and longitude pairs (using the WGS84 datum), specified as two double parameters. However, your API has a `GeoCoordinate` type that can represent coordinates in several coordinate systems, such as Universal Transverse Mercator or Lambert Conformal Conic. You could write an adapter that accepts your `GeoCoordinate` object as a parameter, converts this to geodetic coordinates (latitude, longitude), if necessary, and passes the two doubles to the `MapPlot` API.

4. **Expose a different calling convention for your API**. For example, perhaps you have written a plain C API and you want to provide an object-oriented version of it for C++ users. You could create adapter classes that wrap the C calls into C++ classes. It's open to debate whether this can be strictly considered an adapter pattern, as design patterns are concerned primarily with object-oriented systems, but if you allow some flexibility in your interpretation of the term then you'll see that the concept is the same. The following code gives an example of a C++ adapter for a plain C API. (I'll discuss differences between C and C++ APIs in more detail in the next chapter on styles.)

```
class CppAdapter
{
public:
    CppAdapter()
    {
        mHandle = create_object();
    }
    ~CppAdapter()
    {
        destroy_object(mHandle);
        mHandle = NULL;
    }

    void DoSomething(int value)
    {
        object_do_something(mHandle, value);
    }

private:
    CppAdapter(const CppAdapter &);
    const CppAdapter &operator =(const CppAdapter &);

    CHandle *mHandle;
};
```

### 3.4.3 The Façade Pattern

The Façade design pattern (Figure 3.5) presents a simplified interface for a larger collection of classes. In effect, it defines a higher-level interface that makes the underlying subsystem easier to use. To use Lakos' categorization, the Façade pattern is an example of a multicomponent wrapper (Lakos, 1996). Façade is therefore different from Adapter because Façade simplifies a class structure, whereas Adapter maintains the same class structure.

As your API grows, so can the complexity of using that interface. The Façade pattern is a way to structure your API into subsystems to reduce this complexity and in turn make the API easier to use for the majority of your clients. A façade might provide an improved interface while still allowing access to the underlying subsystems. This is the same as the concept of convenience APIs described in the previous chapter, where additional classes are added to provide aggregated functionality that

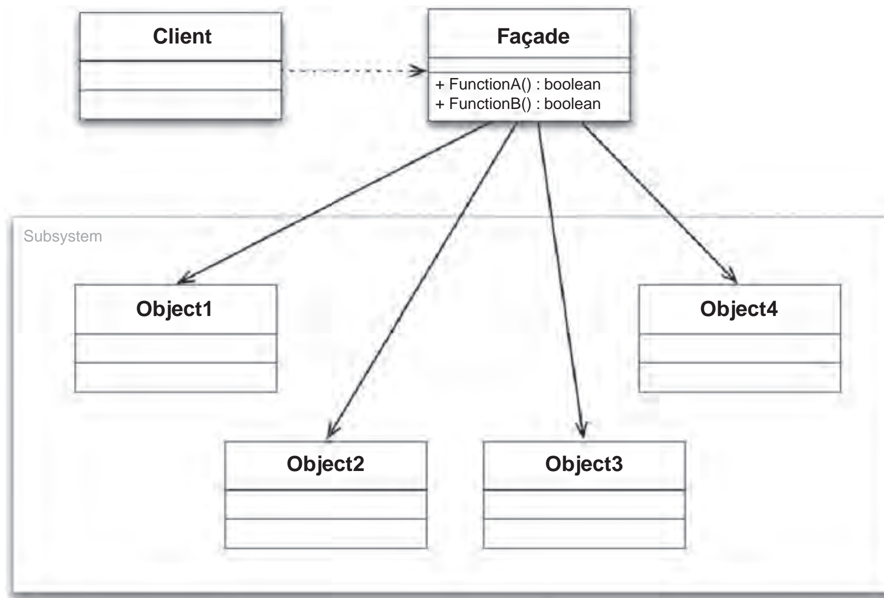


**FIGURE 3.5**

UML diagram of the Façade design pattern.

make simple tasks easy. Alternatively, a façade might completely decouple the underlying subsystems from the public interface so that these are no longer accessible. This is often called an "encapsulating façade."

> **TIP**
>
> A Façade provides a simplified interface to a collection of other classes. In an encapsulating façade, the underlying classes are not accessible.

Let's take a look at an example to illustrate this pattern. Let's assume that you are on holiday and have checked into a hotel. You decide that you want to have dinner and then go to watch a show. To do so, you'll have to call a restaurant to make a dinner reservation, call the theater to book seats, and perhaps also arrange a taxi to pick you up from your hotel. You could express this in C++ as three separate objects that you have to interact with.

```
class Taxi
{
public:
   bool BookTaxi(int npeople, time_t pickup_time);
};
```

```
class Restaurant
{
public:
   bool ReserveTable(int npeople, time_t arrival_time);
};

class Theater
{
public:
   time_t GetShowTime();
   bool ReserveSeats(int npeople, int tier);
};
```

However, let's assume that you're staying in a high-end hotel that has a helpful concierge who can assist you with all of this. In fact, the concierge will be able to find out the time of the show and then, using his local knowledge of the city, work out an appropriate time for your dinner and the best time to order your taxi. Translating this into terms of our C++ design, you now only have to interact with a single object with a far simpler interface.

```
class ConciergeFacade
{
public:
   enum ERestaurant {
      RESTAURANT_YES,
      RESTAURANT_NO
   };
   enum ETaxi {
      TAXI_YES,
      TAXI_NO
   };

   time_t BookShow(int npeople, ERestaurant addRestaurant, ETaxi addTaxi);
};
```

There are various useful applications of the Façade pattern in terms of API design.

1. **Hide legacy code**. Often you have to deal with old, decayed, legacy systems that are brittle to work with and no longer offer a coherent object model. In these cases, it can be easier to create a new set of well-designed APIs that sit on top of the old code. Then all new code can use these new APIs. Once all existing clients have been updated to the new APIs, the legacy code can be completely hidden behind the new façade (making it an encapsulating façade).
2. **Create convenience APIs**. As discussed in the previous chapter, there is often a tension between providing general, flexible routines that provide more power versus simple easy-to-use routines that make the common use cases easy. A façade is a way to address this tension by allowing both to coexist. In essence, a convenience API is a façade. I used the example earlier of the OpenGL library, which provides low-level base routines, and the GLU library, which provides higher-level and easier-to-use routines built on top of the GL library.

3. **Support reduced- or alternate-functionality APIs**. By abstracting away the access to the underlying subsystems, it becomes possible to replace certain subsystems without affecting your client's code. This could be used to swap in stub subsystems to support demonstration or test versions of your API. It could also allow swapping in different functionality, such as using a different 3D rendering engine for a game or using a different image reading library. As a real-world example, the Second Life viewer can be built against the proprietary KDU JPEG-2000 decoder library. However, the open source version of the viewer is built against the slower OpenJPEG library.

## 3.5 OBSERVER PATTERN

It's very common for objects to call methods in other objects. After all, achieving any non-trivial task normally requires several objects collaborating together. However, in order to do this, an object A must know about the existence and interface of an object B in order to call methods on it. The simplest approach to doing this is for `A.cpp` to include `B.h` and then to call methods on that class directly. However, this introduces a compile-time dependency between A and B, forcing the classes to become tightly coupled. As a result, the generality of class A is reduced because it cannot be reused by another system without also pulling in class B. Furthermore, if class A also calls classes C and D, then changes to class A could affect all three of these tightly coupled classes. Additionally, this compile-time coupling means that users cannot dynamically add new dependencies to the system at run time.

> **TIP**
>
> An Observer lets you decouple components and avoid cyclic dependencies.

I will illustrate these problems, and show how the observer pattern helps, with reference to the popular Model–View–Controller (MVC) architecture.

### 3.5.1 Model–View–Controller

The MVC architectural pattern requires the isolation of business logic (the Model) from the user interface (the View), with the Controller receiving user input and coordinating the other two. MVC separation supports the modularization of an application's functionality and offers a number of benefits.

1. Segregation of Model and View components makes it possible to implement several user interfaces that reuse the common business logic core.
2. Duplication of low-level Model code is eliminated across multiple UI implementations.
3. Decoupling of Model and View code results in an improved ability to write unit tests for the core business logic code.
4. Modularity of components allows core logic developers and GUI developers to work simultaneously without affecting the other.

The MVC model was first described in 1987 by Steve Burbeck and Trygve Reenskaug at Xerox PARC and remains a popular architectural pattern in applications and toolkits today. For example, modern UI toolkits such as Nokia's Qt, Apple's Cocoa, Java Swing, and Microsoft's Foundation Class library were all inspired by MVC. Taking the example of a single checkbox button, the current on/off state of the button is stored in the Model, the View draws the current state of the button on the screen, and the Controller updates the Model state and View display when the user clicks on the button.

**TIP**

The MVC architectural pattern promotes the separation of core business logic, or the Model, from the user interface, or View. It also isolates the Controller logic that affects changes in the Model and updates the View.

The implication of MVC separation on code dependency means that View code can call Model code (to discover the latest state and update the UI), but the opposite is not true: Model code should have no compile-time knowledge of View code (because it ties the Model to a single View). Figure 3.6 illustrates this dependency graph.

In a simple application, the Controller can effect changes to the Model based on user input and also communicate those changes to the View so that the UI can be updated. However, in real-world applications the View will normally also need to update to reflect additional changes to the underlying Model. This is necessary because changing one aspect of the Model may cause it to update other dependent Model states. This requires Model code to inform the View layer when state changes happen. However, as already stated, the Model code cannot statically bind and call the View code. This is where observers come in.

The Observer pattern is a specific instance of the Publish/Subscribe, or pub/sub, paradigm. These techniques define a one-to-many dependency between objects such that a publisher object can notify all subscribed objects of any state changes without depending on them directly. The observer pattern

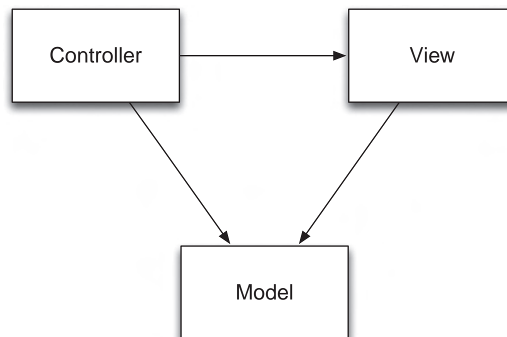


**FIGURE 3.6**

An overview of dependencies in the MVC model. Both the Controller and the View depend on the Model, but Model code has no dependency on Controller code or View code.

is therefore an important technique in terms of API design because it can help you reduce coupling and increase code reuse.

### 3.5.2 Implementing the Observer Pattern

The typical way to implement the observer pattern is to introduce two concepts: the subject and the observer (also referred to as the publisher and subscriber). One or more observers register interest in the subject, and then the subject notifies all registered observers of any state changes. This is illustrated in Figure 3.7.

This can be implemented using base classes to specify the abstract interface for both of these cases, as follows:

```
#include <map>
#include <vector>

class IObserver
{
public:
   virtual ~IObserver() {}
   virtual void Update(int message) = 0;
```

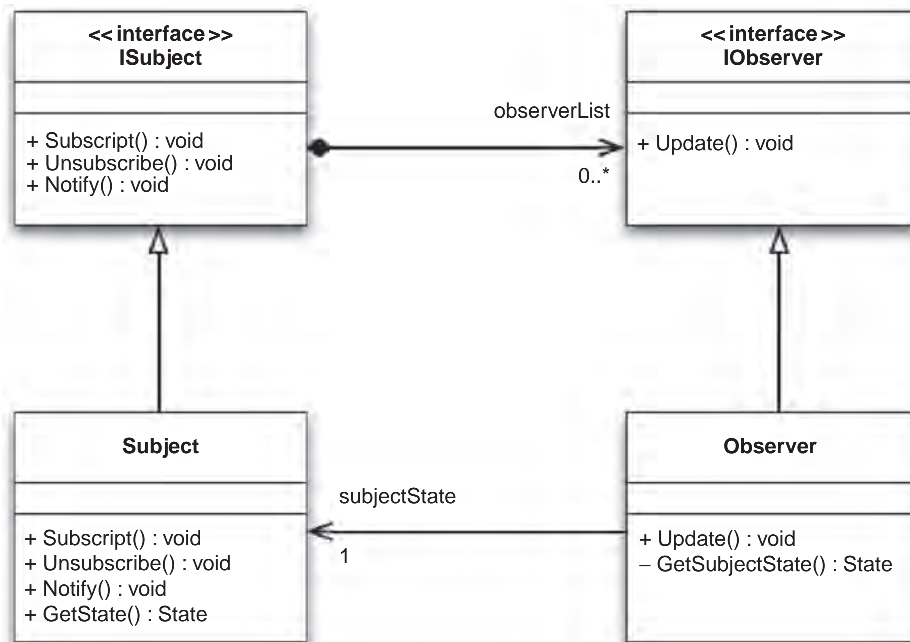


**FIGURE 3.7**

UML representation of the Observer pattern.

```
};

class ISubject
{
public:
   ISubject();
   virtual ~ISubject();
   virtual void Subscribe(int message, IObserver *observer);
   virtual void Unsubscribe(int message, IObserver *observer);
virtual void Notify(int message);

private:
   typedef std::vector<IObserver *> ObserverList;
   typedef std::map<int, ObserverList> ObserverMap;
   ObserverMap mObservers;
};
```

In this design, I've added support for the subject to be able to register and emit notifications for multiple different message types. This allows observers to subscribe to only the specific messages they are interested in. For example, a subject that represents a stack of elements might wish to send out separate notifications when elements are added to or removed from that stack. Using the aforementioned interfaces, you can define a minimal concrete subject class as follows:

```
#include "observer.h"

class MySubject : public ISubject
{
public:
   enum Messages { ADD, REMOVE };
};
```

Finally, you can create observer objects by simply inheriting from the `IObserver` abstract base class and implementing the `Update()` method. The following code demonstrates putting all of these concepts together:

```
#include "subject.h"
#include <iostream>

class MyObserver : public IObserver
{
public:
   explicit MyObserver(const std::string &str) :
      mName(str)
   {}

   void Update(int message)
   {
      std::cout << mName << " Received message ";
      std::cout << message << std::endl;
```

```
    }

private:
    std::string mName;
};

int main(int, char **)
{
    MyObserver observer1("observer1");
    MyObserver observer2("observer2");
    MyObserver observer3("observer3");
    MySubject subject;

    subject.Subscribe(MySubject::ADD, &observer1);
    subject.Subscribe(MySubject::ADD, &observer2);
    subject.Subscribe(MySubject::REMOVE, &observer2);
    subject.Subscribe(MySubject::REMOVE, &observer3);

    subject.Notify(MySubject::ADD);
    subject.Notify(MySubject::REMOVE);

    return 0;
}
```

This example demonstrates creating three separate observer classes and subscribes them for different combinations of the two messages defined by the `MySubject` class. Finally, the calls to `subject.Notify()` cause the subject to traverse its list of observers that have been subscribed for the given message and calls the `Update()` method for each of them. The important point to note is that the `MySubject` class has no compile-time dependency on the `MyObserver` class. The relationship between the two classes is dynamically created at run time.

Of course, there may be a small performance cost for this flexibility—the cost of iterating through a list of observers before making the (virtual) function call. However, this cost is generally insignificant when compared to the benefits of reduced coupling and increased code reuse. Also, as I covered in the previous chapter, you must take care to unsubscribe any observers before you destroy them otherwise the next notification could cause a crash.

### 3.5.3 Push versus Pull Observers

There are many different ways to implement the Observer pattern, with the example I just presented being only one such method. However, I will note two major categories of observers: push-based and pull-based. This categorization determines whether all the information is pushed to an observer via arguments to the `Update()` method or whether the `Update()` method is simply used to send a notification about the occurrence of an event; if the observer wishes to discover more details, then they must query the subject object directly. As an example, a notification that the user has pressed the Return key in a text entry widget may pass the actual text that the user typed as a parameter of

the `Update()` method (push) or it may rely on the observer calling a `GetText()` method on the subject to discover this information if it needs it (pull).

Figure 3.7 illustrates a pull observer pattern because the `Update()` method has no arguments and the observer can query the subject for its current state. This approach allows you to use the same simple `IObserver` for all observers in the system. By comparison, a push-based solution would require you to define different abstract interfaces for each `Update()` method that requires a unique signature. A push-based solution is useful for sending small commonly used pieces of data along with a notification, such as the checkbox on/off state for a checkbox state change. However, it may be inefficient for larger pieces of data, such as sending the entire text every time a user presses a key in a text box widget.