13 Using Design Patterns

Design patterns encode the knowledge of experienced designers. Patterns are design components that can be mapped into code for a particular application. A discussion of patterns in general is beyond the scope of this course; in this chapter, we look at some of the simpler patterns and show how they can be mapped into C++ code.

The "bible" of design patterns is the book *Design Patterns* (Gamma, Helm, Johnson, and Vlissides 1995). The patterns discussed in this chapter are all taken from this book. You should familiarize yourself with at least the GoF[75] patterns, so that you can recognize them when working on an existing code base or talking with experienced designers. Gang of Four

The examples in this chapter may seem to give unnecessarily complicated solutions to trivial problems. They may give you the feeling that patterns are a waste of time. However, patterns are not intended to solve trivial problems. Patterns are useful when similar, but more complex, problems arise in large-scale programming.

You've already seen one well-known design pattern, the ***Iterator***. In fact, by now you probably use it intuitively, without even thinking about it as a complex pattern. Using an iterator offers a number of advantages over traversing a data structure 'manually': With an iterator, you don't have to know the internal representation of the data structure to traverse it (and this representation can change without affecting the traversing clients), you can easily traverse the same structure in parallel (using different iterators), and you can use the same interface (that of the iterator) to traverse a multitude of very different data structures (e.g., list vs. tree vs. set). Another GoF pattern is the ***Façade*** we discussed in the previous lecture (see Section 12.7.2 on page 278). Iterator Façade

13.1 Singleton

Problem: Ensure that a class has only one instance at all times and provide a global point of access to it.

Solution: The problem has two parts: the first part is to ensure that only one instance of a class can exist; the second is to provide users with access to the unique instance. 532

[75] The four authors are often referred to as "The Gang of Four" or "GoF". Their book is sometimes called "The GoF Book".

```
class Singleton
{
public:
    static Singleton & instance();
    void fun();
private:
    int uid;
    static Singleton * pInstance;
    Singleton();
    Singleton(const Singleton &);
    Singleton & operator=(const Singleton &);
    ~Singleton();
};
```

Figure 185: Declaration for class `Singleton`

```
int globalUID = 77;

// Inaccessible pointer to the unique instance.
Singleton * Singleton::pInstance = 0;

// Private constructor.
Singleton::Singleton()
{
    uid = globalUID++;
}

// Public member function allows user to create the unique instance.
Singleton & Singleton::instance()
{
    if (!pInstance)
        pInstance = new Singleton;
    return *pInstance;
}

// Simple test function.
void Singleton::fun()
{
    cout << "I am Singleton " << uid << endl;
}
```

Figure 186: Definitions for class `Singleton`

Figure 185 on the facing page shows a declaration for class `Singleton` based on Alexandrescu's example (Alexandrescu 2001, pages 129–133). Here are some points of interest in this declaration:

- The function `fun` and the data member `uid` are included just to illustrate that the singleton can do something besides merely exist.
- The only way that a user can access the singleton is through the static reference function `instance`.
- Access to the unique instance of the singleton is through the `private`, `static` pointer, `pInstance`.
- The constructor is `private`. The copy constructor and assignment operator are `private` and unimplemented. Thus the singleton cannot be constructed (except for the first time), passed by value, assigned, or otherwise copied.
- The destructor is also `private` and unimplemented. This ensures that the singleton cannot be accidentally deleted. (This could be considered as a memory leak, but that is harmless because there is only one instance anyway.)

533

Figure 186 on the preceding page shows the implementation of class `Singleton`. The constructor is conventional; the body given here merely demonstrates that it works. The function `instance` constructs the singleton when it is called for the first time, and subsequently just returns a reference to it. A user must access the singleton object as `Singleton::instance()`. Any attempt to copy it or pass it by value causes a compile-time error. For the particular singleton defined here, all the user can do is apply the function `fun` to the singleton. Obviously, other functions would be added in a practical application. This is in fact a rather simple version of the Singleton pattern and it is not robust enough for all applications. For details about its improvement, see (Alexandrescu 2001, pages 133–156).

Figure 187 on the following page demonstrates the singleton in action. Running it will result in:

534

535

```
I am Singleton 77
I am Singleton 77
I am Singleton 77
Inside f: I am Singleton 77
```

The first three lines of output show that there is only one instance; if more instances had been created, `globalUID` would have been incremented. The fourth line shows that the singleton has been passed as a reference.

536

13.2 Composite

Problem: Suppose we have a hierarchy of classes. We may also have collections of instances of these classes. A collection might be homogeneous (members all of the same class) or heterogeneous (members of different classes). The problem is to make these collections behave in the same way as elements of the class hierarchy. The collections are called *composite classes* and the pattern that solves the problem is called Composite.

```
void f(Singleton & s)
{
   cout << "Inside f: ";
   s.fun();
}

int main()
{
   Singleton::instance().fun();
   Singleton::instance().fun();
   Singleton::instance().fun();

   // A singleton can be passed by reference.
   f(Singleton::instance());

   // The following lines are all illegal.
   // Singleton s1;
   // Singleton s2 = Singleton::instance();

   return 0;
}
```

Figure 187: A program that tests class `Singleton`

Solution: To implement the Composite pattern, all we have to do is incorporate the composite classes into the class hierarchy. Since putting them into the hierarchy forces them to implement the base class interface, the effect will be to make their behaviour similar to that of other classes in the hierarchy.

The following example is a highly simplified typesetting application. The abstract base class `Text` has two pure virtual functions: `set`, which "typesets" text to an output stream provided as a parameter and `size`, which returns the size of a text. Initially, there are three derived classes, 537 `Blank`, `Character`, and `Word`, as shown in Figure 188 on the next page.

538 Suppose that we want to typeset paragraphs. Paragraphs are related to the classes we have in 539 that they should be able to implement the method `set`, but they are also different in that a paragraph has many words or characters and is typically typeset between margins. Nevertheless, 540 we can add a class `Paragraph` to the hierarchy, as shown in Figure 189 on page 288.

Class `Paragraph` has a parameter in the constructor giving the width in which to set the paragraph. (A more realistic example would also allow this value to be set after construction.) We provide a function `addElem` that allows us to build paragraphs out of text elements (words, characters, and whatever else might be added later). These text elements are stored in a STL `vector`. The important points of this pattern are that class `Paragraph` inherits from class `Text` and implements `set` and `size`, making it a part of the hierarchy, while at the same time aggregating elements of its own superclass, `Text`.

541 Figure 190 on page 288 shows the implementation of the `Paragraph` function `set`. It uses a `stringstream` to store the text elements that make up a line. When there is not enough space

```
class Text
{
public:
   virtual void set(ostream & os) const = 0;
   virtual size_t size() const = 0;
};

class Blank : public Text
{
public:
   Blank() {}
   void set(ostream & os) const { os << ' '; }
   size_t size() const { return 1; }
};

class Character : public Text
{
public:
   Character(char c) : c(c) {}
   void set(std::ostream & os) const { os << c; }
   char getChar() const { return c; }
   size_t size() const { return 1; }
private:
   char c;
};

class Word : public Text
{
public:
   Word(const std::string & w) : w(w) {}
   void set(std::ostream & os) const { os << w; }
   string getWord() const { return w; }
   size_t size() const { return w.size(); }
private:
   string w;
};
```

Figure 188: Declarations for the text class hierarchy

on the line for the next text element (and the blank that precedes it), the string stream buffer is filled to `width` with blanks and written to the output stream.

Note that with this Composite pattern, we can now add a new text element to the hierarchy (e.g., an in-line graphic type) and the `set` function of a `Paragraph` will still work, since each new type must still implement `Text`'s functions `set` and `size`. Note that composite structures can be nested: A paragraph can have another paragraph as an element.

```
class Paragraph : public Text
{
public:
   Paragraph(int width) : width(width) {}
   void set(std::ostream & os) const;
   void addElem(Text* e) { elems.push_back(e); }
   size_t size() const {
      ostringstream oss;
      this->set(oss);
      return oss.str().size();
   }
private:
   vector<Text*> elems;
   size_t width;
};
```

Figure 189: A class for paragraphs

```
void Paragraph::set(ostream & os) const
{
   Blank b;
   ostringstream oss;
   for (vector<Text*>::const_iterator it = elems.begin();
                                      it != elems.end(); ++it)
   {
      if (size_t(oss.tellp()) + 1 + (*it)->size() > width)
      {
         while (size_t(oss.tellp()) < width)
            b.set(oss);
         os << oss.str() << endl;
         oss.str("");
         oss.clear();
      }
      if (size_t(oss.tellp()) > 0)
         b.set(oss);
      (*it)->set(oss);
   }
   os << oss.str() << endl;
}
```

Figure 190: Typesetting a `Paragraph`

13.3 Visitor

542

Problem: Figure 191 shows a grid of classes and operations that extend the `Text` hierarchy of the previous section.

	Blank	Char	Word	Paragraph	Chapter
set	⋆	⋆	⋆	⋆	⋆
cap	⋆	⋆	⋆	⋆	⋆
print	⋆	⋆	⋆	⋆	⋆
count	⋆	⋆	⋆	⋆	⋆

Figure 191: A grid of classes and operations

Note that a column corresponds to a class (that implements each operation) and a row corresponds to an operation (that must be implemented in each class). Each ⋆ indicates an action that must be implemented. Here are two ways of organizing the code that implements the operations:

1. In pre-object-oriented days, each operation was implemented as a single function with a big `switch` statement: see Figure 192. This made it easy to add a function, because all of the code would be in one place, but hard to add a class, because this would involve adding a clause to many different functions. 543

2. In an object-oriented language, we put the classes into a hierarchy; the root class of the hierarchy defines default or null versions of each function; and each class implements the functions in its own way. This design has two consequences:

   a) The code becomes fragmented: a function implements one operation in one class.

```
void set(....)
{
   switch(kind)
   {
   case BLANK:
      ....
      break;
   case CHAR:
      ....
      break;
   case WORD:
      ....
      break;
   ....
   }
}
```

Figure 192: Choosing operations using `switch-case`

b) It is easy to add a class, with an implementation for each operation, but hard to add an operation, because each class has to be modified.

c) If the objects are stored in a data structure, it is likely that each function will be responsible for traversing its part of the data structure.

For example, each time we add an operation to a class hierarchy such as `Text`, we have to add one function to each class in the hierarchy. Furthermore, every function we add to class `Paragraph` will iterate over the words in the paragraph.

The problem is to combine these two modes: we would like to retain the fragmented code, which is easier for maintenance, but we would like to organize it by operation rather than by class. We would also like to keep the organization of the data structure out of the individual functions.

Solution: The idea of the Visitor pattern is to define a "visitor" class that knows how to process each kind of object in the hierarchy. The visitor class is an abstract base class that cannot actually do anything useful, but it has derived classes for performing specific operations.

The following steps are required to support visitors in the `Text` hierarchy.

544

1. Create a base class `Visitor`: see Figure 193. This class has a virtual "visiting" function corresponding to each derived class in the `Text` hierarchy. Each visiting function is passed a pointer to an object of the corresponding type in the `Text` hierarchy.

545 546

2. Add functions to "accept" visitors in the `Text` hierarchy, starting with a pure virtual function in the base class, as shown in Figure 194 on page 292. Most of these functions are fairly trivial. For example:

```
void Blank::accept(Visitor & vis) { vis.visitBlank(this); }
void Character::accept(Visitor & vis) { vis.visitCharacter(this); }
```

For composite classes, function `accept` passes the visitor to each component individually:

```
void Paragraph::accept(Visitor & vis)
{
    for (  vector<Text*>::iterator it = elems.begin();
           it != elems.end(); ++it)
       (*it)->accept(vis);
}
```

```
class Visitor
{
public:
   virtual void visitBlank(Blank*) = 0;
   virtual void visitCharacter(Character*) = 0;
   virtual void visitWord(Word*) = 0;
   virtual void visitParagraph(Paragraph*) = 0;
};
```

Figure 193: The base class of the `Visitor` hierarchy

This completes the framework for visiting.

3. The next step is to construct an actual visitor. We will reimplement the typesetting function, `set`, as a visitor. For this, we need a class `setVisitor` derived from `Visitor`: see Figure 195 on the following page. 547

4. Finally, we implement the member functions of `setVisitor`, as in Figure 196 on the next page. It is not necessary to provide a body for `setVisitor::visitParagraph` because it is never called: when a `Paragraph` object accepts a visitor, it simply sends the visitor to each of its elements (see `Paragraph::accept` above). However, we do have to provide a trivial implementation in order to make `Paragraph` non-abstract. (Alternatively, we could have defined a trivial default function in the base class.) 548

5. To typeset a paragraph `p`, we call just: 549

```
p.accept(setVisitor());
```

To demonstrate the flexibility of the Visitor pattern, we can define another visitor that sets the same text in capital letters. The new class is called `capVisitor`: 550

```
class capVisitor : public Visitor
{
public:
   void visitBlank(Blank*);
   void visitCharacter(Character * pc);
   void visitWord(Word*);
   void visitParagraph(Paragraph*);
};
```

Its member functions are the same as those of `setVisitor` except for `visitCharacter` and `visitWord`: 551

```
void capVisitor::visitCharacter(Character * pc)
{
   char ch = pc->getChar();
   cout << toupper(ch);
}

void capVisitor::visitWord(Word * pw)
{
   string word = pw->getWord();
   for (string::iterator it = word.begin(); it != word.end(); ++it)
      *it = toupper(*it);
   cout << ' ' << word;
}
```

To invoke `capVisitor`, we simply construct an instance:

```
p.accept(capVisitor());
```

```
class Text
{
public:
   virtual void accept(Visitor & v) = 0;
   ....
};
```

Figure 194: Adding `accept` to the base class of the `Text` hierarchy

```
class setVisitor : public Visitor
{
public:
    void visitBlank(Blank*);
    void visitCharacter(Character * pc);
    void visitWord(Word*);
    void visitParagraph(Paragraph*);
};
```

Figure 195: A class for typesetting derived from `Visitor`

```
void setVisitor::visitBlank(Blank*)
{
    cout << ' ';
}

void setVisitor::visitCharacter(Character * pc)
{
    cout << pc->getChar();
}

void setVisitor::visitWord(Word * pw)
{
    cout << ' ' << pw->getWord();
}

void setVisitor::visitParagraph(Paragraph * pp)
{
   cout << "I should never be called";
}
```

Figure 196: Implementation of class `setVisitor`

For a single task (typesetting), setting up the Visitor classes seems rather elaborate. Suppose, however, that there were many operations to be performed on the `Text` hierarchy. Once we have defined the `accept` functions, we do not need to make any further changes to that hierarchy. Instead, for each new operation that we need, we derive a class from `Visitor` and define its "visit" functions for each kind of `Text`.

The visitor pattern has some disadvantages:

- If we use a separate function for each operation (like `set` in the original example), we can pass information around simply by adding parameters. Since the operations are independent, each can have its own set of parameters.

  The operations in the Visitor version, however, must all use the protocol imposed by `accept`. This means that they all get exactly the same parameters (none, in our example).

- The structure of composite objects is hard-wired into their `accept` functions. For example, `Paragraph::accept` iterates through its vector of `Text` elements. This makes it difficult to modify the traversal in any way.

  In the Composite example above, `Paragraph::set` inserted line breaks whenever the length of a line would otherwise have exceeded `width`. It is more difficult to provide this behaviour with the Visitor pattern, because `Paragraph::accept` does a simple traversal over the elements and does not provide for any additional actions. Nor is it possible to pass an argument to `Paragraph::accept` giving the position on the current line.

The Visitor pattern implements a form of ***double dispatch***. In C++, like in most object-oriented languages, the function to call depends on only two things: the *receiving object* and the *method* in that object. With double dispatch, the function to execute also depends on the *caller* (or sender) of said message.

double dispatch

13.4 State

Problem: An object has various states and behaves in different ways depending on its state. We could write a `switch` statement to capture this property, as shown in Figure 197 on the following page. This solution is unsatisfactory because any changes – modification of a state's behaviour, adding or removing states, etc. – requires changes to this statement.

552 553

Solution: The State pattern is often implemented by inheritance and dynamic binding. The base class, which may be abstract, defines the action corresponding to a default state or no action at all. Each derived class defines the action for a particular state. The following example demonstrates the State pattern with a scanner whose state determines the text that it recognizes.

The abstract base class `Scanner` shown in Figure 198 on page 295 specifies that any scanner object must accept a character pointer and return a `string`. Note that the pointer is passed by reference, because a scanner will advance the pointer over the buffer and must return its new value to the caller. The result of scanning is the string representing the token scanned. For example, a number scanner might return the string `"123.456"`. Figure 198 on page 295 also shows derived classes that scan white space, numbers, and identifiers.

554 555 556

The scanners may be selected in various ways. Figure 199 on page 296 shows a simple managing

557 558 559

class. An instance of `ScanManager` has a pointer to each kind of scanner; its constructor creates the corresponding objects dynamically and its destructor deletes them. The function `ScanManager::scan` is given a pointer to a character array and it uses the character to choose a scanner object. Figure 200 on page 297 shows a simple example of the scanner in use, resulting in:

560

```
Scan succeeded: <123> <456> <Pirate456> <789> <anotherID>
```

Remarks:

- The nice thing about the State pattern is that it localizes the behaviour corresponding to a given state. There are many simple classes rather than a possibly huge `switch` statement or equivalent. It is easy to modify a state without looking at, or even knowing about, other states. Adding a state is also easy and does not require interfering with existing code.
- In this example, there is a single controlling object, the `ScanManager`, that handles all the state changes. This makes the State pattern look a bit pointless. However, there are other ways of changing state:
  - Each derived class might be responsible for choosing its successor state. A drawback of this approach is that each derived class has to know about one or more other derived classes, which goes against the usual practice of keeping derived classes independent of each other.

```
class StateChanger
{
public:
   void act()
   {
      switch (state)
      {
      case RUNNING:
         // ....
         break;
      case STUCK:
         // ....
         break;
      case BROKEN:
         // ....
         break;
      }
   }
private:
   enum { RUNNING, STUCK, BROKEN } state;
};
```

Figure 197: A class with various states

```
class Scanner
{
public:
   virtual string scan(char * & p) = 0;
};

class ScanBlanks : public Scanner
{
public:
   string scan(char * & p)
   {
      while (isspace(*p))
         p++;
      return string();
   }
};

class ScanNumber : public Scanner
{
public:
   string scan(char * & p)
   {
      string num;
      while (isdigit(*p))
         num += *p++;
      if (*p == '.')
      {
         num += *p++;
         while (isdigit(*p))
            num += *p++;
      }
      return num;
   }
};

class ScanIdentifier : public Scanner
{
public:
   string scan(char * & p)
   {
      string id(1, *p++);
      while (isalpha(*p) || isdigit(*p))
         id += *p++;
      return id;
   }
};
```

Figure 198: State pattern: scanners for white space, numbers, and identifiers

```
class ScanManager
{
public:
   ScanManager();
   ~ScanManager();
   string scan(char *buffer);
private:
   Scanner *ps;
   Scanner *psBlanks;
   Scanner *psNumber;
   Scanner *psIdentifier;
};

ScanManager::ScanManager() :
   ps(0),
   psBlanks(new ScanBlanks()),
   psNumber(new ScanNumber()),
   psIdentifier(new ScanIdentifier())
{ }

ScanManager::~ScanManager()
{
   delete psBlanks;
   delete psNumber;
   delete psIdentifier;
}

string ScanManager::scan(char *buffer)
{
   string result;
   char *p = buffer;
   while (*p != '\0')
   {
      if (isspace(*p))
         ps = psBlanks;
      else if (isdigit(*p))
         ps = psNumber;
      else if (isalpha(*p))
         ps = psIdentifier;
      else
         throw "illegal character.";
      string token = ps->scan(p);
      if (token.length() > 0)
         result += "<" + token + "> ";
   }
   return result;
}
```

Figure 199: State pattern: the scanner controller class

```
int main()
{
   ScanManager sm;
   char *test = "123 456 Pirate456\t789 anotherID ";
   try
   {
      cout << "Scan succeeded: " << sm.scan(test).c_str() << endl;
   }
   catch (const char *error)
   {
      cerr << "Scan failed: " << error << endl;
   }
   return 0;
}
```

Figure 200: State pattern: using the scanner

  - The state could be chosen externally. In the example above, `ScanManager` could provide another member function that was told what kind of token to expect and could set the state accordingly.

- It would be nice to put some useful data into the base class. In C++, we cannot do this efficiently, because the scanning is performed by separate objects that cannot access data in the base object. There are several workarounds for this problem:
  - We could construct state objects dynamically when needed, pass them the information they need, and delete them at the next state transition. This is clearly less efficient than the solution above, but the inefficiency might be acceptable if state transitions were infrequent.
  - We could provide additional functions for updating the `Scanner` objects before executing them. But it is not obvious when to call such functions.
  - Some object oriented languages (e.g., Self) provide *dynamic inheritance*, which allows an object to move around the class hierarchy, adopting the behaviour of the class it belongs to at any given time.

13.5 Bridge

Problem: Tight coupling between an interface and an implementation make it difficult to change either part without affecting the other.

Solution: The Bridge pattern separates an abstraction from its implementation so that the two can vary independently. It is also known as the ***Handle/Body*** pattern. A Bridge is a kind of generalized Pimpl (see Section 12.6 on page 267).

Handle/Body

There is an example of the Bridge pattern (called Handle/Body) in Koenig and Moo (Koenig and Moo 2000). In Section 13.4 (pages 243–245), there is a declaration of class `Student_info` in which the only data member is `Core *cp`. The pointer `cp` points an instance of either the base class `Core` or the derived class `Grad`: see Figure 202 on the facing page. An instance of class `Student_info` is constructed by reading data from a file. The class of `*cp` is determined by a character in the file: see Figure 201.

561
562
563

```
istream& Student_info::read(istream& is)
{
    delete cp;          // delete previous object, if any

    char ch;
    is >> ch;           // get record type

    if (ch == 'U') {
        cp = new Core(is);
    } else {
        cp = new Grad(is);
    }

    return is;
}
```

Figure 201: Setting the pointer (Koenig & Moo, page 245)

```
#ifndef GUARD_Student_info_h
#define GUARD_Student_info_h

#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>

#include "Core.h"

class Student_info {
public:
    // constructors and copy control
    Student_info(): cp(0) { }
    Student_info(std::istream& is): cp(0) { read(is); }
    Student_info(const Student_info&);
    Student_info& operator=(const Student_info&);
    ~Student_info() { delete cp; }

    // operations
    std::istream& read(std::istream&);

    std::string name() const {
        if (cp) return cp->name();
        else throw std::runtime_error("uninitialized Student");
    }
    double grade() const {
        if (cp) return cp->grade();
        else throw std::runtime_error("uninitialized Student");
    }

    static bool compare(const Student_info& s1,
                        const Student_info& s2) {
        return s1.name() < s2.name();
    }

private:
    Core* cp;
};

#endif
```

Figure 202: Separating interface and implementation (Koenig & Moo, pages 243–244)

13.6 Command

The Command pattern *"encapsulates a request as an object, thereby enabling you to parameterize clients with different requests, queue or log requests, and support undoable operations"* (Gamma, Helm, Johnson, and Vlissides 1995, page 233). At the appropriate time, a request is activated in some way. The Gang of Four use a function called `Execute` to activate a request, but `operator()` provides a slightly neater solution.

564 The `Command` shown in Figure 203 class stores requests to print messages. The message to be printed is passed as an argument to the constructor. To activate a command `c`, the user writes `c()`. For example, commands can be stored in a vector and then activated sequentially, as shown
565 in Figure 204.

```
class Command
{
public:
    Command(string message) : message(message) {}
    void operator()() { cout << message << endl; }
private:
    string message;
};
```

Figure 203: A simple command class

```
vector<Command> commands;
commands.push_back(Command("First message"));
commands.push_back(Command("Second message"));
// ....
// a long time later:
for (   vector<Command>::iterator it = commands.begin();
        it != commands.end();
        ++it )
    (*it)();
```

Figure 204: Using the command class

Remember that you can declare `operator()` with any number of parameters (see Section 6.7.2 on page 121).

13.7 Observer

The Observer pattern has a *Subject*, which is an object that changes its state, and one or more
Model/View *Observers* that respond to changes in the Subject's state. Other names for Subject/Observer are Publish/Subscribe, and Model/View (in which case there may be a Controller as well).

566 In Figure 205 on the next page, the Subject is a thermometer `th` that registers a temperature. The Observers are a `DigitalReader dr` and an `AnalogReader ar`, which display the temperature

in different ways. Each Observer has a pointer to its Subject, set when the Observer is created. The Subject must also be told about the Observers: this is done by the call `th.add()`.

When the system has been set up, the temperature changes are recorded by calling `th.setTemp()`. The thermometer notifies each of the observers of a change in temperature, and they display the new temperature. If a reader is removed from the thermometer, it stops displaying. Figure 206 shows the output generated by the program of Figure 205.

567

```
#include "thermometer.h"
#include "digital.h"
#include "analog.h"

using namespace std;

int main()
{
   Thermometer th;
   DigitalReader dr(&th);
   AnalogReader ar(&th);
   th.add(&dr);
   th.add(&ar);
   th.setTemp(5);
   th.setTemp(10);
   th.remove(&dr);
   th.setTemp(15);
   return 0;
}
```

Figure 205: `main.cpp`

```
New temperature: 5
Digital: 5
Analog:  -----

New temperature: 10
Digital: 10
Analog:  ----------

New temperature: 15
Analog:  ---------------
```

Figure 206: Output from the program of Figure 205

Rather than working with thermometers and readers directly, we create a framework for the Observer pattern with abstract base classes. Figure 207 on the following page shows the base class for a Subject. It has functions to add and remove Observers, and a function that enables the Subject to notify its Observers about updates. The Observers are stored in a list.

568

569

```
#include <list>

class Observer;

class Subject
{
public:
   virtual ~Subject() {}
   void add(Observer *obs);
   void remove(Observer *obs);
   void notify();
private:
   std::list<Observer*> observers;
};
```

Figure 207: `subject.h`

Figure 208 on the next page shows the implementation of the base class. Functions `add()` and `remove()` use library functions to insert and remove entries from the list of Observers, and `notify()` sends a message to each Observer in the list. Note that the Subject passes a pointer
570 to itself when notifying an Observer.

Figure 209 on the facing page shows the class `Thermometer`, which is derived from `Subject`. This class has functions related to the specific kind of Subject: in this case, functions that get and set the temperature. `setTemp()` is called by some external agency to indicate that the temperature has changed. `getTemp()` is called by the Observers to obtain the current temperature. (The example is over-simplified: a real Subject would probably do more than just provide get and set capabilities.)

571 The implementation of `Thermometer`, shown in Figure 210 on page 304, is straightforward. The
572 main point to note is that `setTemp()` calls `notify()`, which is inherited from the base class.

Figure 211 on page 304 shows the abstract base class for an Observer. The only required method is `notify()`, which must be implemented by derived classes. Since no code is needed for the base class, there is no implementation file for `Observer`.

573 In this example, we provide two kinds of Observer. Figure 212 on page 304 shows the header file
574 for class `DigitalReader` and Figure 213 on page 305 shows the corresponding implementation file. The main point to note is that `notify()` checks that the Subject passed to it is the Subject to which the `DigitalReader` owns a pointer.

It might seem simpler for `notify()` to use the pointer passed to it, by calling `changed->getTemp()`. However, this would not work because `changed` has type `Subject*`, and the base class `Subject` does not have a function `getTemp()`.

Is the test '`changed == th`' really necessary? To answer this, we have to consider circumstances under which it might be false. There might be a complex network of Subjects and Observers, and Observers might be added indiscriminately to Subjects. In such a context, it makes sense for the Observer to check that the notifier is indeed one that it is interested in. The point being illustrated here is that an Observer may choose to ignore notifications that are of no interest to

```
#include "subject.h"
#include "observer.h"

using namespace std;

void Subject::add(Observer *obs)
{
   observers.push_back(obs);
}

void Subject::remove(Observer *obs)
{
   observers.remove(obs);
}

void Subject::notify()
{
   for (  list<Observer*>::iterator it = observers.begin();
          it != observers.end();
          ++it)
   {
      (*it)->notify(this);
   }
}
```

Figure 208: `subject.cpp`

```
#include "subject.h"

class Thermometer : public Subject
{
public:
   Thermometer() : temp(0) {}
   int getTemp();
   void setTemp(int newTemp);
private:
   int temp;
};
```

Figure 209: `thermometer.h`

```
#include "thermometer.h"

#include <iostream>
using namespace std;

int Thermometer::getTemp()
{
   return temp;
}

void Thermometer::setTemp(int newTemp)
{
   temp = newTemp;
   cout << "\nNew temperature: " << newTemp << endl;
   notify();
}
```

Figure 210: `thermometer.cpp`

```
class Subject;

class Observer
{
public:
   virtual ~Observer() {}
   virtual void notify(Subject *changed) = 0;
};
```

Figure 211: `observer.h`

it. In a more realistic example, the condition '`changed == th`' might be replaced by another condition to test the relevance of the update.

```
#include "observer.h"
#include "thermometer.h"

class DigitalReader : public Observer
{
public:
   DigitalReader(Thermometer *th);
   void notify(Subject *changed);
private:
   Thermometer *th;
};
```

Figure 212: `digital.h`

```
#include "digital.h"

#include <iostream>
using namespace std;

DigitalReader::DigitalReader(Thermometer *th) : th(th)
{}

void DigitalReader::notify(Subject *changed)
{
   if (changed == th)
      cout << "Digital: " << th->getTemp() << endl;
}
```

Figure 213: `digital.cpp`

The second kind of Observer is an `AnalogReader`, shown as a header file (**??**igure fig-obs-9) and an implementation file (Figure 215 on the next page). To make things more interesting, class `AnalogReader` is derived from class `AnalogDisplay`. This provides an example of ***multiple implementation inheritance***, in which a derived class has two base classes (see Section 8.5.2 on page 175). 575 576

Apart from the multiple inheritance, `AnalogReader` is similar to `DigitalReader`. Its `notify()` function checks that it is interested in the Observer, and then calls function `draw()` from `AnalogDisplay` to display the thermometer reading.

```
#include "observer.h"
#include "thermometer.h"

class AnalogDisplay
{
public:
   void draw(int val);
};

class AnalogReader : public AnalogDisplay, public Observer
{
public:
   AnalogReader(Thermometer *th);
   void notify(Subject *changed);
private:
   Thermometer *th;
};
```

Figure 214: `analog.h`

```
#include "analog.h"

#include <iostream>
using namespace std;

void AnalogDisplay::draw(int val)
{
   cout << "Analog:  ";
   for (int i = 0; i < val; ++i)
      cout << '-';
   cout << endl;
}

AnalogReader::AnalogReader(Thermometer *th) : th(th) {}

void AnalogReader::notify(Subject *changed)
{
   if (changed == th)
      draw(th->getTemp());
}
```

Figure 215: analog.cpp

13.8 Pattern Languages

Despite its age, the original book on *Design Patterns* (Gamma, Helm, Johnson, and Vlissides 1995) is still the reference to read. The code examples in this book are mostly in C++ (with some in Smalltalk) – this is probably the main reason this book is usually not given to beginner Java programmers. The other reason is that, at the time it was written, patterns were a topic for advanced, professional software developers, not learners of object-oriented programming. In style and addressed audience, the Java based book (Freeman and Freeman 2004) is probably the polar opposite.

Alexander's pattern language

But the original idea of patterns and pattern languages comes from building architecture. Christopher Alexander, together with colleagues, examined re-occurring problems in design and construction and how they could be solved in a generic fashion. Their solution was the development of a ***pattern language***, where (Alexander, Ishikawa, and Silverstein 1977):

> *"Each pattern describes a problem that occurs over and over again in our environment, and then describes the core solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice."*

The book is very accessible to non-architects, which is perhaps one of the reasons it inspired design patterns and pattern languages in software engineering.

References

Alexander, C., S. Ishikawa, and M. Silverstein (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.

Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Generic Patterns Applied*. Addison-Wesley.

Freeman, E. and E. Freeman (2004). *Head First Design Patterns*. O'Reilly.

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

Koenig, A. and B. E. Moo (2000). *Accelerated C++: Practical Programming by Example*. Addison-Wesley.