Chapter 5
Elementary Design Patterns

As one may expect, a software engineer who has had experience developing a number of application systems is able to utilise the expertise gained in future projects. Although two applications may not be alike and may exhibit relatively little similarity at the outset, delving deeper into the design may reveal a number of similar issues. Working on a variety of projects, a software engineer gets exposure to problems that are common to multiple scenarios, which hones his/her ability to identify repeated instances of problems and spell out solutions for them fairly quickly. From an object-oriented perspective, what it means is that two different applications may provide design issues that are alike; the solutions may involve the development of a set of classes with similar functionalities and relationships. Thus the class structures for the two subproblems may end up being the same although there may be differences in details.

An example from the imperative paradigm may help the reader better understand the above discussion. Consider two applications, one a university course registration system and the other a human resource (HR) system for some organisation. In the first example we may wish to provide screens which allow a student to register for classes that can be selected from a list. Let us say that we will list courses sorted according to the departments in the university and that within the department, the courses will be listed in ascending order of course identifiers—this information is to be retrieved from disk before it can be displayed. In the second application, let us assume that we want to retrieve employee-related information from disk and print the information in the sorted order of departments, and within each department in the ascending order of employee names. Although the applications are quite different, the scenarios have similarity: both involve reading information which is data related to some application from disk and then sorting the data based on some fields in it. An efficient sorting mechanism should be used in both cases. We could envisage similar processing in many other applications as well. A professional who has some experience in application design and is conversant with such scenarios should be

B. Dathan and S. Ramnath, *Object-Oriented Analysis, Design and Implementation*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-24280-4_5

able to identify the proper approach to be taken for solving the problem and employ it effectively.

In object-oriented systems, we break up the system into objects and develop classes that serve as blueprints for creating objects. Therefore, unlike the imperative world where we need to recognise the appropriate algorithms for solving a problem, the task in object-oriented systems is to recognise the necessary classes, interfaces and relationships between them for solving a specific design problem. Such an approach, which can then be tailored to solve similar design problems that recur in a multitude of applications, is called a **design pattern**.

Here are some quotes from the literature:

> Design patterns are partial solutions to common problems such as separating an interface from a number of alternate implementations, wrapping around a set of legacy classes, protecting a caller from changes associated with specific problems. A design pattern is composed of a small number of classes that, through delegation and inheritance, provide a robust and modifiable solution. These classes can be adapted and refined for the specific system under construction [1].
>
> A pattern is a way of doing something, or a way of pursuing an intent [2].

A number of design patterns are known, and as one may expect, they vary in the level of difficulty of comprehending and employing them. In this chapter we study three design patterns. Although the patterns we treat here are relatively simple, they also are quite popular and useful. So the reader is likely to find them being utilised in applications and may use them often in his/her own code.

In Sect. 5.2, we study the Iterator pattern which helps us traverse a collection with no regard to the way the collection is organised. The second pattern, Singleton, is discussed in Sect. 5.3. This pattern is used when it is known that we should have exactly one instance of a certain class. The main utility of this pattern is thus in its ability to support data integrity. Finally, we study the Adapter pattern which helps us develop new classes that satisfy an interface by exploiting the functionality of the existing classes.

5.1 Iterator

In many applications we need to maintain collections which are objects that store other objects. For example, a telephone company system could have a collection object that stores an object for each of its customers; an airline system is likely to maintain information about each of its flights and the references to them may be stored in a collection object. Depending on the type of application, the actual data structure employed may differ. In Chap. 3 we talked about collections in general, and in Chap. 4 we discussed collection classes such as `java.util.LinkedList`.

Popular data structures that implement collections include linked lists, queues, stacks, double-ended queues, binary search trees, B-Trees and hash tables.

Let us imagine a collection implemented as a list that stores instances of type `Object`. The list provides several methods for accessing the elements including the following:

1. `size()`, which returns the number of elements in the list.
2. `get(int index)`, which returns the element at a specific position given by `index`.

Consider a client that maintains a list of `Objects` as below:

```
private ListImplementation1 elements;
```

If the client needs to process all of the objects in the collection, it needs to set up a loop to access every element.

```
for (int index = 0; index < elements.size(); index++) {
  Object object = elements.get(index);
  // process object
}
```

Assume that after the system development we determine that an alternate implementation of the collection is warranted. The client code is modified so that the elements are a list of type `ListImplementation2`:

```
private ListImplementation2 elements;
```

Suppose that `ListImplementation2` does not support either of the above two methods, `size()` and `get(int index)`. Instead, the supported operations include:

1. `reset()`: makes the collection ready to return elements.
2. `next()`: returns an element from the collection in no specific order. Every element is returned exactly once. The method returns `null` if there are no more elements.

Obviously, the client code that iterated using `size` and `get(int index)` need to be rewritten. One way to iterate would be:

```
Object object;
for (elements.reset(), object = elements.next(); object != null;
                       object = elements.next()) {
  // process object
}
```

This requires modification of code within the client, which is not very desirable. Although changes are inevitable in most systems, alterations in implementation of a subsystem should not necessitate modifications of other subsystems. In other words, the system should be loosely coupled. Otherwise, the cost of maintenance can be high.

In fact, the above cost can be completely avoided if we ensure that interfaces supported by classes never change. In the example we have been discussing, this means that `ListImplementation1` and `ListImplementation2` both support a common set of methods.

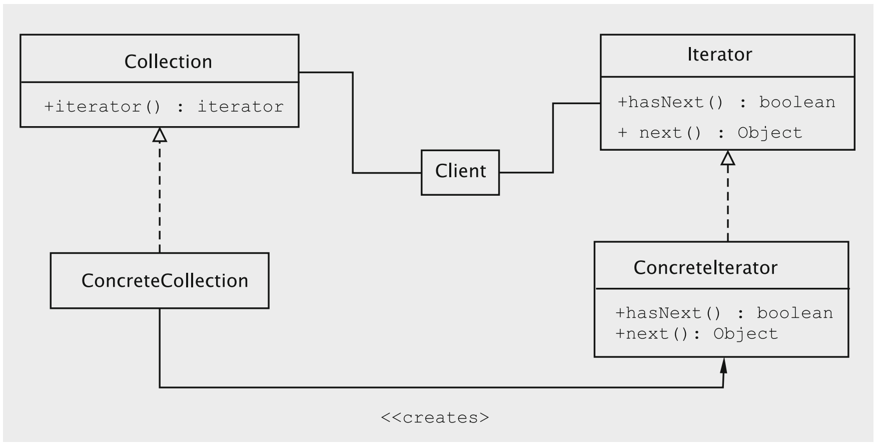

Fig. 5.1 Iterator structure

Another way to ensure less coupling between the client and the collection class would be to require that collection traversal be implemented by employing a special type of object which provides a standard way of iterating over the elements, independent of the internal organisation of the collection. Every collection is then required to return an **iterator** object, which provides these standard methods to traverse the collection.

For example, if `myCollection` refers to an object of type `Collection`, the expression

```
myCollection.iterator()
```

returns an iterator object.

The iterator supports a method called `next()`, which returns an element from the collection each time it is called. No element is returned more than once and if enough calls are made, all elements will be returned. The caller may check whether all elements have been returned by using the method `hasNext()`, which returns `true` if not all elements have been returned.

Thus, in our scheme, we have the following classes and interfaces as shown in Fig. 5.1.

1. `Collection`, an interface that allows the usual operations to add and delete objects, plus the method `iterator()` that returns an iterator object.
2. `Iterator`, an interface that supports the operations `hasNext()` and `next()` described earlier.
3. Implementation of the `Collection` interface: obviously, every implementation must implement the `iterator` method by creating an `Iterator` object and returning it.
4. Implementation of the `Iterator` interface: this class must cooperate with the code in (3) above to properly access and return the elements of the collection.
5. Client code that uses the collection.

Let us look at the class `LinkedList` in Java, which implements `Collection` and supports the `iterator` method.

```
Collection collection = new LinkedList();
collection.add("Element 1");
collection.add(new Integer(2));
for (Iterator iterator = collection.iterator(); iterator.hasNext(); ) {
  System.out.println(iterator.next());
}
```

The first line creates a `LinkedList` object whose reference is stored in the variable `collection`. We add two elements to the collection, a `String` object and an `Integer` object. Every object of type `Collection` supports the `iterator` method, and this method is invoked in the initialisation of the `for` loop. The returned object `iterator` is of the type `Iterator`. Before entering the loop the first time or in any succeeding iteration, we make sure that we have not processed all the elements. The method call `iterator.hasNext()` returns `true` if there is at least one element in the collection not yet retrieved since the `iterator` was created. Such a collection element is retrieved in the body of the loop by the call `iterator.next()`. In this code, we simply print the elements. Thus, we will end up printing `Element 1` and `2` in successive lines.

Changes are inevitable in almost all applications, so we must ensure that these changes do not have widespread ramifications. If every collection class implements the `iterator` method that returns an object of type `Iterator`, clients can use the iterator object to traverse the collection making the process independent of the collection implementation. This insulates the client code from changes in the collection class.

One natural question that may arise in this context is the following: *why is it necessary to return an iterator?* One could argue that it is enough to ensure that every collection supports the methods `hasNext` and `next`. This argument has some validity, but the drawback of this approach is that the design and implementation of the collection class itself becomes more complicated. In addition to managing the elements in the collection, the collection class will have to keep track of every client that navigates the elements. This results in the design being less cohesive. As we shall see in the implementation below, the iterator pattern provides a clean solution to this by separating each traversal process from the collection itself.

5.1.1 Iterator Implementation

In this section we describe how to implement an iterator in Java. Suppose we have the interface `Queue`, which allows adding and removing of objects using the queue discipline (FIFO).

```
public interface Queue {
  public boolean add(Object value);
  public Object remove();
}
```

We implement the above interface in `LinkedQueue`. The inner class `Node` stores an object and the reference to the next element in the linked list. The head and tail of the Queue are stored in the variables `head` and `tail` respectively.

```
import java.util.*;
public class LinkedQueue implements Queue {
  private Node head;
  private Node tail;
  private int numberOfElements;
  private class Node {
    private Object data;
    private Node next;
    private Node(Object object, Node next) {
      this.data = object;
      this.next = next;
    }
    public Object getData() {
      return data;
    }
    public void setNext(Node next) {
      this.next = next;
    }
    public Node getNext() {
      return next;
    }
  }
  // Queue methods
}
```

The add method creates an instance of `Node` and inserts it at the tail of the list. The code is straightforward.

```
  public boolean add(Object value) {
    Node node = new Node(value, null);
    if (tail == null) {
      tail = head = node;
    }
    else {
      tail.setNext(node);
      tail = node;
    }
    numberOfElements++;
    return true;
  }
```

The `remove` method also employs the standard approach to deleting from a queue. Before changing the value of `head`, we retrieve the contents of the first node in the queue so we can return the deleted element.

```
  public Object remove() {
    if (head == null) {
      return null;
    }
    Object value = head.getData();
    head = head.getNext();
    if (head == null) {
      tail = null;
    }
```

```
        numberOfElements--;
        return value;
      }
    // The iterator method returns a new Iterator.
      public Iterator iterator() {
        return new QueueIterator();
      }
```

The iterator is implemented as an inner class. In the interface `java.util.Iterator`, there are three methods: `hasNext`, `next`, and `remove`, the last operation being optional. The `Iterator` object must maintain the list of elements in the queue that are not yet returned to the client. For this we take advantage of the fact that the `LinkedQueue` class itself has a linked list and that list is accessible from the code within `QueueIterator`. However, the iterator class must not modify the field `head` in `LinkedQueue`; for this, we maintain a field called `cursor` within `QueueIterator`. This field is initialised to `head` when the iterator object is created.

```
private class QueueIterator implements Iterator {
  private Node cursor;
  public QueueIterator() {
    cursor = head;
  }
  // hasNext, next, and remove
}
```

Our plan is to return the elements as they appear in the queue. Therefore, the code for `hasNext` is quite simple: we just need to make sure that `cursor` is not null. Hence, we have

```
  public boolean hasNext() {
    return cursor != null;
  }
```

To retrieve the next element, we must first make sure that there is at least one element not supplied to the client. That is, `hasNext()` does not return a `null` value. Then, we just move one element forward by setting `cursor` to `cursor.getNext()`.

```
public Object next() {
    if (!hasNext()) {
      return null;
    }
    Object object = cursor.getData();
    cursor = cursor.getNext();
    return object;
}
```

Finally, the implementation of the `remove` method is the simplest of all because we decided not to support this functionality! As a result, the method body is empty.

```
public void remove() {
}
```

The above implementation shows the clean separation between the collection and the iterator. Another advantage of this approach is that we incur no additional complexity if there are multiple iterators being employed simultaneously, as the following code illustrates.

```
Collection collection = new LinkedList();
collection.add(new Integer(1));
collection.add(new Integer(2));
for (Iterator iterator1 = collection.iterator(); iterator1.hasNext(); ) {
  Integer int1 = (Integer) iterator1.next();
  int count = 0;
  for (Iterator iterator2 = collection.iterator(); iterator2.hasNext(); ) {
    Integer int2 = (Integer) iterator2.next();
    if (int1.equals(int2)) {
      count++;
    }
  }
  System.out.println(int1 + count);
}
```

5.2 Singleton

As a second example of a scenario that repeats across applications, we note that in many situations we want to ensure that there is just one object of a certain class. For example, although a computer system may have many printers, there is usually only one spooler. A company has only one president. A single-processor system obviously can have only one CPU.

To create a class that can only be instantiated once, we note that the constructor cannot have the `public` access specifier. Instead, we provide a method called `instance()` that returns the only instance of the class.

```
public class B {
  private static B singleton;
  private B() {
  }
  public static B instance() {
    if (singleton == null) {
      singleton = new B();
    }
    return singleton;
  }
  // application code
}
```

The major observation to be made here is that to get the only instance of class `B`, a client invokes the static method `instance`. This is because the constructor is private, so the code from outside the class cannot instantiate `B`. When the class is loaded, the field `singleton` will be set to `null`. In the very first call to `instance`, an instance of `B` is created and the reference stored in `singleton`. Further calls to `instance` result in no new allocations, and the value in `singleton` is returned.

Notice some of the other major features of the implementation:

1. Clients need not maintain a variable to keep track of the instance. Simply by invoking the static method `instance`, the instance can be retrieved.
2. The class can be subclassed. The subclasses themselves may be singletons.
3. Instead of using a singleton, one may have a class with static methods. But since static methods are not virtual, subclassing will not be able to override these methods.

5.2.1 Subclassing Singletons

In some applications it is necessary to develop subclasses of a singleton class where the subclasses themselves are singletons. For an example of such a system, consider a distributed system with one or more server machines and many client sites. A server machine runs several server processes. In our example, we have exactly four processes.

1. A general-purpose server that provides many services including time, directory, file, replication and name services. However, some of these services are somewhat primitive in nature.
2. A directory server that provides sophisticated directory service.
3. A file server that allows reading and updating of data.
4. A file server that allows only reading; only new files can be written.

Since the general-purpose server already provides the basic support for directory and file management, it seems reasonable to assume that the specialised classes for instantiating the directory and file servers are subclasses of the class for the general-purpose server. All the classes are singletons.

For a second example, consider a large corporation with offices all around the world. The corporate headquarters is located in, say, New York. Every country in which the corporation operates has its own separate national headquarters to control operations within that country. For instance, the company may operate in France and have its headquarters in Paris. A sample hierarchy is given in Fig. 5.2.

Let us further assume that the functionality of each of the national headquarters is quite similar to the functionality of the corporate headquarters. However, there are differences between the corporate headquarters and individual national headquarters (in matters such as labour and other laws, currency, etc.).

Thus, we implement the above system using a singleton class for the corporate headquarters and a separate singleton subclass for each of the national headquarters.

In general, the problem of interest in this context boils down to the following: We need to implement two classes, `B` and `D` where `B` is the superclass of `D`, and both classes are singletons.

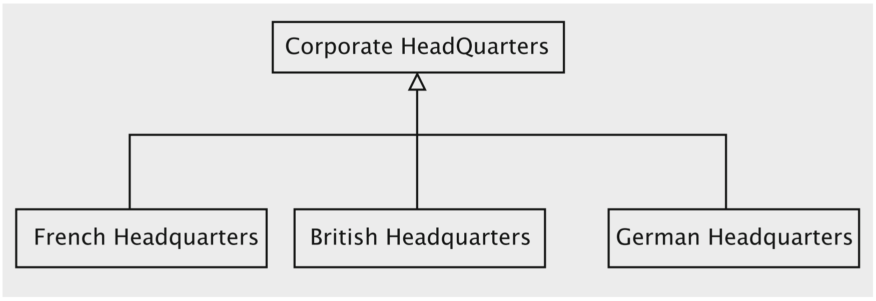

Fig. 5.2 Singleton hierarchy

Consider the implementation of `B` as we had it in Sect. 5.2. Suppose we attempt to implement `D` as below.

```
public class D extends B {
  private static D singleton;
  private D() {
  }
  public static D instance() {
    if (singleton == null) {
      singleton = new D();
    }
    return singleton;
  }
  // application code
}
```

This code has a problem: since `B` has a private constructor, it is impossible for `D` to be instantiated. The constructor of `D` makes an implicit call to the no-argument constructor of the superclass, `B`, and the compiler blocks this because the superclass's constructor is private.

The solution developed below recognises the fact that the instantiation of `B` has to be done differently when we have a singleton hierarchy.

1. `B` is instantiated through the `instance` method. The class does not have any public constructors.
2. For `D` to be instantiated, it is necessary that some constructor of `B` be accessible from the code within `D`. Since this constructor cannot be public, it follows that the constructor be protected. Therefore, we have

```
public class B {
  private static B singleton;
  protected B() {
  }
  public static B instance() {
    if (singleton == null) {
      singleton = new B();
    }
    return singleton;
  }
```

```
    // more application code
  }
  public class D extends B {
    private static D singleton;
    protected D() {
    }
    public static D instance() {
      if (singleton == null) {
        singleton = new D();
      }
      return singleton;
    }
    // more application code
  }
```

3. The code has the flaw that the code within class `D` can instantiate multiple instances of `B`, violating the fundamental property of a singleton class.
   Therefore, we must control the behavior when `B's` constructor is invoked from `D`. This can achieved by using the Java reflection mechanism, which, as we saw earlier, allows Java code to discover the properties and behaviour of an object at the execution time. In particular, this mechanism allows, at runtime, the discovery of the name of the class to which an object belongs, the names of the supported interfaces, field names, methods and constructors. Let `C` be a class and `p` a reference created as below.

```
C p = new C();
```

Since the expression `p.getClass().getName()` returns `'C'`, we can modify the class `B` as below.

```
import java.lang.reflect.*;
public class B {
  private static B singleton;
  protected B() {
    if (getClass().getName().equals("B")) {
      throw new Exception();
    }
  }
  public static B instance() {
    if (singleton == null) {
      singleton = new B();
    }
    return singleton;
  }
  // more application code
}
```

Any attempt to instantiate `B` directly will now fail because the invocation will have to go through the protected method, which throws an exception whenever `B` is instantiated. Our solution requires that the constructor knows what kind of object is being created at the execution time by calling for RTTI, which, in this case, is obtained through reflection. In this situation, the `instanceof` operator does not suffice; every instance of `D` is an instance of `B` and the resulting constructor would not allow the creation of any object whatsoever.

4. The above modification introduces the problem that instances of B cannot be created at all! (When the `instance()` method of B invokes the constructor, an exception is thrown.) This is corrected by introducing a private constructor. Since constructors must have differing signatures, we introduce an artificial parameter to this constructor. This step thus yields

```
import java.lang.reflect.*;
public class B {
  private static B singleton;
  protected B() throws Exception {
    if (getClass().getName().equals("B")) {
      throw new Exception();
    }
  }
  private B(int i) {
  }
  public static B instance() {
    if (singleton == null) {
      singleton = new B(1);
    }
    return singleton;
  }
  // more application code
}
```

The descendants of B use the protected constructor, but only to create instances of B that are embedded in instances of the descendants, which cannot be independently accessed. Only one explicitly constructed instance of B exists, which is done using the private constructor.

5.3 Adapter

Suppose that during the design stage of a piece of software we formalise an interface, i.e., come up with a set of methods that we want implemented. Assume that we have available to us a class whose application programming interface (API)—the set of methods available to clients—is similar to the demands we have identified, but still does not quite match the interface we arrived at. Rather than implement the interface completely from scratch, which may entail considerable expenditure in terms of time and money, we may be better off by tweaking the existing class. However, modifying the class directly to arrive at the new functionality is also not the best approach for two fairly obvious reasons:

1. We need to understand the details of the implementation of the given class, which may be expensive.
2. Future changes to the original class to fix bugs, enhance functionality, etc., will not be available in the interface's implementation.

Therefore, we need a better strategy for this problem.

Before discussing a better solution, let us specify the problem a little more formally. We have a class C that supports a set, say, M_C, of methods. We assume that we need to implement interface I that contains a set, M_I, of methods. By some measure, let us say that M_I resembles M_C, but the methods in the two sets are not quite the same. The problem is to figure out the best way to arrive at an implementation for the interface I given the fact that there are similarities between the methods in M_I and M_C.

This is a problem that frequently occurs in practice. As toolsmiths, it is important for us not to start from scratch nor delve into other's ventures that require an inordinate investment of time and money, if at all possible. The strategy that we have in mind is to develop a class A that implements I, whereby each method in M_I is realised by a combination of calls to a subset of the methods in M_C.

The approach outlined above is known as the adapter pattern. Its main function is to adapt an existing module to implement a given application interface. For obvious reasons, it promotes code reuse.

The structure of the pattern is shown in Fig. 5.3. The interface `Client Interface` corresponds to the interface I in the above discussion, and the client wants to invoke methods in this interface. For this purpose, the client maintains a reference to an `Adapter` instance, which implements the methods in `ClientInterface`. Notice that `method1` and `method2` form the set M_I in our earlier discussion. The class `Adaptee` is an existing class (C in our discussion) and the set of methods formed by `adapteeMethod1` and `adapteeMethod2` corresponds to the set M_C.

In our strategy, the adapter creates and maintains a reference to an adaptee instance. Now, suppose the client wants to invoke method `method1` in `ClientInterface`. The adapter satisfies the request by using the methods of the adaptee.

As an example, suppose we are given the interface `Deque`. A `Deque` instance is a collection of objects in which elements can be added and deleted at either end. Moreover, the interface also supports methods to peek at the head and tail of the

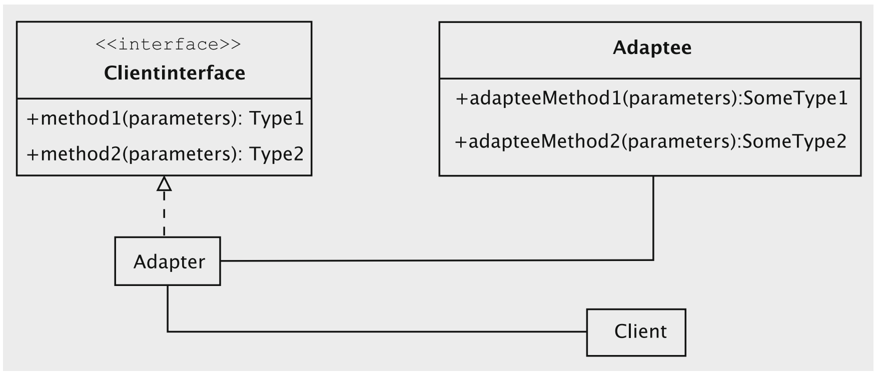
<<interface>>
Clientinterface
+method1(parameters): Type1
+method2(parameters): Type2
Adaptee
+adapteeMethod1(parameters):SomeType1
+adapteeMethod2(parameters):SomeType2
Adapter
Client

Fig. 5.3 Object adapter structure

collection (`getElementAtHead` and `getElementAtTail`), determine the size (`size`), delete all elements (`clear`), and return an iterator (`iterator`).

```
import java.util.*;
public interface Deque {
  public boolean addAtTail(Object value);
  public Object removeElementAtTail();
  public Object getElementAtTail();
  public boolean addAtHead(Object value);
  public Object removeElementAtHead();
  public Object getElementAtHead();
  public int size();
  public void clear();
  public Iterator iterator();
}
```

In Java, we have the class `LinkedList` in which elements can be added, deleted, or peeked at any position: (`add(int index, E element)`, `remove(int index)`, and `get(int index)`); the size can be determined (`size`), all elements can be deleted(`clear`) and, an iterator on the collection can be retrieved (`iterator`). However, there are two disadvantages to using the `LinkedList` class in place of a `Deque` implementation.

1. In some cases, the method names are different from the ones in the `Deque` interface.
2. The class is more general than the demands of the `Deque` interface. For instance, the `remove(int index)` method can be used to delete an element at any position, not just at the head or tail. This violates the `Deque` discipline.

Nonetheless, a subset of the `LinkedList` class methods have enough similarity with the methods of `Deque` that we can use the former in the interface's implementation. Let us assume that `Deque` is implemented in a class named `DequeImpl`.

The adapter pattern comes in handy for the purpose. There are two forms of the pattern; **object adapter**s and **class adapter**s. In this example, we use an object adapter. An object adapter creates an adapter class that implements a given interface using an instance of an existing class, which is the adaptee. In the scenario we just described above, the interface is `Deque`, the adaptee is an instance of `LinkedList`, and the adapter is `DequeImpl`. The adapter creates and maintains a reference to an adaptee object and, of course, implements all of the methods the interface. Methods of the interface are implemented by delegating the work to the adaptee object.

Thus, the class `DequeImpl` would be structured as below.

```
import java.util.*;
public class DequeImpl implements Deque {
  private List list =  new LinkedList();
  // methods as dictated by Deque
}
```

The idea is that the object `list` will be used to store the deque.

Let us now look at some of the methods. When a request to add at the tail comes in, we simply insert it at the tail of the `List` object. This is done by invoking the `add` method in `List`. Thus, the code for `addAtTail` is

```
public boolean addAtTail(Object value) {
  return list.add(value);
}
```

Similarly, removing from the tail is accomplished by invoking the `remove` method in `List` as below.

```
public Object removeElementAtTail() {
  if (list.size() > 0) {
    return list.remove(list.size() - 1);
  }
  return null;
}
```

Notice that we need to protect the code; so, we must ensure that the list contains at least one element before invoking the remove operation.

The code for accessing the tail element is

```
public Object getElementAtTail() {
  if (list.size() > 0) {
     return list.get(list.size() - 1);
  }
  return null;
}
```

The methods for processing the head element are similar.

Methods for getting the size, iterator and clearing the `Deque` object are quite simple.

```
public int size() {
  return list.size();
}
public void clear() {
  list.clear();
}
public Iterator iterator() {
  return list.iterator();
}
```

The `equals` method can be implemented by comparing the `Deque` object with another object, element by element.

```
public boolean equals(Object object) {
  Deque other = (Deque) object;
  if (other.size() != this.size()) {
    return false;
  }
  Iterator thisIterator = this.iterator();
  Iterator otherIterator = other.iterator();
  while (thisIterator.hasNext() && otherIterator.hasNext()) {
    if (!(thisIterator.next().equals(otherIterator.next()))) {
      return false;
    }
  }
  return true;
}
```

Note that in the above example we keep a reference to the `List` object within `Deque` rather than extend an implementation of `List`. We are thus adapting the `List` object, and hence the pattern is called an object adapter. The methods of `List` are unavailable to the user of `DequeImpl`.

In contrast, we could have extended `LinkedList` and called the methods of the superclass to carry out the actions of the `Deque` interface. Such an adapter is called a **class adapter**. This is not as flexible as the object-based approach because we are extending a specific class and that decision is made at compile time. The object adapter has the advantage that the choice of the adaptee class can be postponed until execution time. Moreover, in the case of the class adapter, all of the public methods of the extended class are exposed to the client. The downside to an object adapter is that it introduces one more object into the system.

5.4 Discussion and Further Reading

A major goal of employing design patterns is to cater to the changes that may become necessary during the lifetime of a system. Changes are inevitable in any application system and systems must be designed so that they can handle changes in specifications with minimum fuss: any specification change should result in the modification of a small number of classes with no wide ramifications within the system. An implementation based on a design that cannot accommodate changes very well is likely to have a short life or will be too expensive to maintain.

Using design patterns can also help in the understanding of designs more quickly because they are well-understood solutions to frequently occurring problems. For instance, if we say that a certain part of the system is built using the adapter pattern, we can immediately understand how classes and interfaces in that part of the system are organised.

Although several design patterns are quite easy to understand, there are some that are quite difficult. Regardless of the difficulty, most design patterns use a combination of some of the following approaches.

1. Program to a type. If at all possible, commit to a class as late as possible. This allows us to use the appropriate implementation at execution time. Since implementations can change during a system's lifetime, this strategy helps to ensure that we are adapting to changes as they occur. For example, in the following code we define `mySet` as of type `Set` rather than as `HashSet` or `TreeSet`, which are implementations.

```
Set mySet;
// code
mySet = new HashSet();
```

2. To make the above point feasible, ensure that the specifications are spelled out using interfaces.

3. Use composition and inheritance appropriately. When it is required that we inherit the type and implementation of a specific class, use inheritance. In many situations, however, we can get around this requirement and use composition.
4. Isolate what can vary and encapsulate it. Define a suitable interface for the varying entity. The code in the rest of the system can then use the idea in (1) above to refer to the actual object that implements the interface. If changes require creating a new class for the interface, the code that references the old object can easily switch to an instance of the new class.

The reader should look for the above principles while studying design patterns.

Understanding design patterns is a relatively easy task compared to identifying situations where these patterns are applicable. A first step toward meeting this challenging task can be taken by a thorough understanding of the patterns (study good examples) and convincing oneself of the fact that the ideas used are indeed useful. After that a little bit of experience in using the patterns in a few situations should make the process simpler. Once again, it appears that patterns that are simpler to understand are also easier to apply.

The best source of reference for design patterns is the classic catalog of the patterns by Gamma, Helm, Johnson, and Vlissides (aka Gang of Four, abbreviated GoF) *Design Patterns: Elements of Reusable Object-Oriented Software* [3]. This was the first book that talked about the fundamental patterns (23 of them). A number of other books [4–6] that explain the patterns are also available in the market, but the GoF book remains unmatched for its elegance and precision.

Projects

1. The following interface `DateInterface` contains a subset of the methods in the class `java.util.Date`. We have indicated what the methods do mostly by quoting from the documentation in Sun's JDK. (For more details of what the methods do, please see the JDK documentation.)

```
public interface DateInterface {
  // Returns the year minus 1900
  public int getYear();
  // Sets the year
  public void setYear(int year);
  /* Returns the month represented by this date. The value is
between 0 and 11. */
  public int getMonth();
  // sets the month
  public void setMonth(int month);
  // returns the day of the month
  public int getDate();
  // sets the day of the month
  public void setDate(int date);
  //  Returns the day of the week
  public int getDay();
  // Returns the hour between 0 and 23
  public int getHours();
  // Sets the hour
  public void setHours(int hours);
  //  Returns the number of minutes past the hour
```

```
    public int getMinutes();
    // Sets the minutes of this Date object
    public void setMinutes(int minutes);
    //  Returns the number of seconds past the minute
    public int getSeconds();
    //  Sets the seconds of this Date object
    public void setSeconds(int seconds);
    /*  Returns the number of milliseconds since January 1, 1970,
  00:00:00 GMT */
    // represented by this Date object.
    public long getTime();
    //  Sets this Date object to represent a point in time that is time
    //  milliseconds after January 1, 1970 00:00:00 GMT.
    public void setTime(long time);
  }
```

Your task is to implement the above interface using the adapter pattern. For this, locate a class other than `java.util.Date` to be used as the adaptee. Implement some suitable constructors as well.

2. Study the class `java.util.StringTokenizer`. Implement the following interface, `PushbackTokenizer,` as a class adaptor with `StringTokenizer` as the adaptee.

```
  public interface PushbackTokenizer {
  // Returns the next token
    public String nextToken();
  // Returns true if and only if there are more tokens
    public boolean hasMoreTokens();
  /* The token read is pushed back, so it can be read again
  using nextToken.*/
    public void pushback();
  }
```

5.5 Exercises

1. The interface `java.util.Iterator` contains an additional method `remove()`. Study what this method does and explain any difficulties that you forsee if this is implemented.
2. Implement a list class that implements the following interface:

```
    // add at the tail
    public void add(Object object);
    // add at the given position
    public void add(Object object, int index);
    // delete and return the element at the given position;
    // return null if no such element exists
    public Object delete(int index);
    // return the number of elements in the list.
    public int size();
    // return an object of type java.util.ListIterator
    public ListIterator listIterator();
```

3. Look up Java documentation for details on the `clone` method. Suppose that a singleton class implements the `clone()` method. How does it affect the integrity of the system? Discuss how you may circumvent these difficulties.
4. We have already noted that the singleton pattern can be realised by having a class that contains nothing but a set of static methods. Find a real-life example of a singleton class and show that this observation is true. Next, identify a pair of classes in which one is a subclass of the other and both are singletons. Attempt to use the 'static methods approach' to make them singletons and convince yourself of the difficulties.
5. Identify singleton classes in a university that maintains several separate collections including the following for storing the list of faculty members, the list of students, the list of staff members, and one that maintains a list of these collections themselves.
6. Compare and contrast the interfaces `Enumeration` and `Iterator` in `java.util`.
7. Suppose that we would like to implement a Java interface using the class adapter pattern. However, exposing some methods of the adaptee could result in loss of integrity. Suggest a way to hide such methods.
8. What are the proper methods for a `Stack` object? With this background, examine the design of the `java.util.Stack` class and see if it the design is sound.

References

1. B. Bruegge, A.H. Dutoit, *Object-Oriented Software Engineering* (Prentice Hall, New Jersey, 2000)
2. B. Goetz, Java theory and practice: Be a good (event) listener. guidelines for writing and supporting event listeners, http://www.ibm.com/developerworks/, July 2005
3. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, Boston, 1994)
4. S.J. Metsker, *Design Patterns Java Workbook* (Addison-Wesley, Boston, 2002)
5. A. Shalloway, J.R. Trott. *Design Patterns Explained A New Perspective on Object-Oriented Design* (Addison-Wesley, Boston, 2004)
6. E. Freeman, E. Robson, B. Bates, K. Sierra, *Head First Design Patterns (Head First)* (O'Reilly, California, 2004)