

# Design Patterns

6.170 Lectures 14–16

October 16, 18, and 19, 2000

## Contents

<b>1</b>	<b>Design patterns</b>	<b>137</b>
1.1	Examples	137
1.2	When (not) to use design patterns	138
1.3	Why should you care?	138
<b>2</b>	<b>Creational patterns</b>	<b>139</b>
2.1	Factories	139
2.1.1	Factory method	140
2.1.2	Factory object	141
2.1.3	Prototype	143
2.2	Sharing	145
2.2.1	Singleton	145
2.2.2	Interning	146
2.2.3	Flyweight	148
<b>3</b>	<b>Structural patterns</b>	<b>151</b>
3.1	Wrappers	151
3.1.1	Adapter	151
3.1.2	Decorator	153
3.1.3	Proxy	154
3.1.4	Subclasing vs. delegation	155
3.2	Composite	156
<b>4</b>	<b>Behavioral patterns</b>	<b>158</b>
4.1	Multi-way communication	158
4.1.1	Observer	158
4.1.2	Blackboard	160
4.1.3	Mediator	160
4.2	Traversing composites	160
4.2.1	Interpreter	162
4.2.2	Procedural	163
4.2.3	Visitor	164
4.3	State	166

Reading: Chapter 15 of *Program Development in Java* by Barbara Liskov

# 1 Design patterns

A design pattern is:

- a standard solution to a common programming problem
- a technique for making code more flexible by making it meet certain criteria
- a design or implementation structure that achieves a particular purpose
- a high-level programming idiom
- shorthand for describing certain aspects of program organization
- connections among program components
- the shape of an object diagram or object model

## 1.1 Examples

Here are some examples of design patterns which you have already seen. For each design pattern, this list notes the problem it is trying to solve, the solution that the design pattern supplies, and any disadvantages associated with the design pattern. A software designer must trade off the advantages against the disadvantages when deciding whether to use a design pattern. Tradeoffs between flexibility and performance are common, as you will often discover in computer science (and other fields).

### Encapsulation (data hiding)

**Problem:** Exposed fields can be directly manipulated from outside, leading to violations of the representation invariant or undesirable dependences that prevent changing the implementation.

**Solution:** Hide some components, permitting only stylized access to the object.

**Disadvantages:** The interface may not (efficiently) provide all desired operations. Indirection may reduce performance.

### Subclassing (inheritance)

**Problem:** Similar abstractions have similar members (fields and methods). Repeating these is tedious, error-prone, and a maintenance headache.

**Solution:** Inherit default members from a superclass; select the correct implementation via run-time dispatching.

**Disadvantages:** Code for a class is spread out, potentially reducing understandability. Run-time dispatching introduces overhead.

### Iteration

**Problem:** Clients that wish to access all members of a collection must perform a specialized traversal for each data structure. This introduces undesirable dependences and does not extend to other collections.

**Solution:** Implementations, which have knowledge of the representation, perform traversals and do bookkeeping. The results are communicated to clients via a standard interface.

**Disadvantages:** Iteration order is fixed by the implementation and not under the control of the client.

### Exceptions

**Problem:** Errors occurring in one part of the code should often be handled elsewhere. Code should not be cluttered with error-handling code, nor return values preempted by error codes.

**Solution:** Introduce language structures for throwing and catching exceptions.

**Disadvantages:** Code may still be cluttered. It can be hard to know where an exception will be handled. Programmers may be tempted to use exceptions for normal control flow, which is confusing and usually inefficient.

These particular design patterns are so important that they are built into Java. Other design patterns are so important that they are built into other languages. Some design patterns may never be built into languages, but are still useful in their place.

## 1.2 When (not) to use design patterns

The first rule of design patterns is the same as the first rule of optimization: delay. Just as you shouldn't optimize prematurely, don't use design patterns prematurely. It may be best to first implement something and ensure that it works, then use the design pattern to improve weaknesses; this is especially true if you do not yet grasp all the details of the design. (If you fully understand the domain and problem, it may make sense to use design patterns from the start, just as it makes sense to use a more efficient rather than a less efficient algorithm from the very beginning in some applications.)

Design patterns may increase or decrease the understandability of a design or implementation. They can decrease understandability by adding indirection or increasing the amount of code. They can increase understandability by improving modularity, better separating concerns, and easing description. Once you learn the vocabulary of design patterns, you will be able to communicate more precisely and rapidly with other people who know the vocabulary. It's much better to say, "This is an instance of the visitor pattern" than "This is some code that traverses a structure and makes callbacks, and some certain methods must be present, and they are called in this particular way and in this particular order."

Most people use design patterns when they notice a problem with their design — something that ought to be easy isn't — or their implementation — such as performance. Examine the offending design or code. What are its problems, and what compromises does it make? What would you like to do that is presently too hard? Then, check a design pattern reference. Look for patterns that address the issues you are concerned with.

The canonical design pattern reference is the "gang of four" book, *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995. Design patterns are popular now, so new books continue to appear.

## 1.3 Why should you care?

If you are a talented designer and programmer, or you have a lot of time to gain experience, you may encounter or invent many design patterns yourself. However, this is not an effective use of your time. A design pattern represents work by someone else who also encountered the problem, tried many possible solutions, and selected and described one of the best. You should take advantage of that.

Design patterns may seem abstract at first, or you may not be convinced that they address a significant problem. You will come to appreciate them as you build and modify larger systems — perhaps during your work on MapQuick and the Gizmoball final project.

## 2 Creational patterns

### 2.1 Factories

Suppose you are writing a class to represent a bicycle race. A race consists of many bicycles (among other objects, perhaps).

```
class Race {  
  
    Race createRace() {  
        Frame frame1 = new Frame();  
        Wheel frontWheel1 = new Wheel();  
        Wheel rearWheel1 = new Wheel();  
        Bicycle bike1 = new Bicycle(frame1, frontWheel1, rearWheel1);  
        Frame frame2 = new Frame();  
        Wheel frontWheel2 = new Wheel();  
        Wheel rearWheel2 = new Wheel();  
        Bicycle bike2 = new Bicycle(frame2, frontWheel2, rearWheel2);  
        ...  
    }  
  
}
```

You can specialize Race for other bicycle races:

```
// French race  
class TourDeFrance extends Race {  
  
    Race createRace() {  
        Frame frame1 = new RacingFrame();  
        Wheel frontWheel1 = new Wheel700c();  
        Wheel rearWheel1 = new Wheel700c();  
        Bicycle bike1 = new Bicycle(frame1, frontWheel1, rearWheel1);  
        Frame frame2 = new RacingFrame();  
        Wheel frontWheel2 = new Wheel700c();  
        Wheel rearWheel2 = new Wheel700c();  
        Bicycle bike2 = new Bicycle(frame2, frontWheel2, rearWheel2);  
        ...  
    }  
  
    ...  
}
```

```
// all-terrain bicycle race  
class Cyclocross extends Race {  
  
    Race createRace() {  
        Frame frame1 = new MountainFrame();  
        Wheel frontWheel1 = new Wheel27in();  
    }  
}
```

```

    Wheel rearWheel1 = new Wheel27in();
    Bicycle bike1 = new Bicycle(frame1, frontWheel1, rearWheel1);
    Frame frame2 = new MountainFrame();
    Wheel frontWheel2 = new Wheel27in();
    Wheel rearWheel2 = new Wheel27in();
    Bicycle bike2 = new Bicycle(frame2, frontWheel2, rearWheel2);
    ...
}

...
}

```

In the subclasses, `createRace` returns a `Race` because the Java compiler enforces that overridden methods have identical return types.

For brevity, the code fragments above omit many other methods relating to bicycle races, some of which appear in each class and others of which appear only in certain classes.

The repeated code is tedious, and in particular, we weren't able to reuse method `Race.createRace` at all. (There is a separate issue of abstracting out the creation of a single bicycle to a function; we will use that without further discussion, as it is obvious, at least after 6.001.) There must be a better way. The Factory design patterns provide an answer.

### 2.1.1 Factory method

A factory method is a method that manufactures objects of a particular type.

We can add factory methods to `Race`:

```

class Race {

    Frame createFrame() { return new Frame(); }
    Wheel createWheel() { return new Wheel(); }
    Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
        return new Bicycle(frame, front, rear);
    }
    // return a complete bicycle without needing any arguments
    Bicycle completeBicycle() {
        Frame frame = createFrame();
        Wheel frontWheel = createWheel();
        Wheel rearWheel = createWheel();
        return createBicycle(frame, frontWheel, rearWheel);
    }

    Race createRace() {
        Bicycle bike1 = completeBicycle();
        Bicycle bike2 = completeBicycle();
        ...
    }
}

```

Now subclasses can reuse `createRace` and even `completeBicycle` without change:

```
// French race
class TourDeFrance extends Race {

    Frame createFrame() { return new RacingFrame(); }
    Wheel createWheel() { return new Wheel700c(); }
    Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
        return new RacingBicycle(frame, front, rear);
    }
}

class Cyclocross extends Race {

    Frame createFrame() { return new MountainFrame(); }
    Wheel createWheel() { return new Wheel26inch(); }
    Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
        return new RacingBicycle(frame, front, rear);
    }
}
```

The `create...` methods are called *factory methods*.

### 2.1.2 Factory object

If there are many objects to construct, including the factory methods in each class can bloat the code and make it hard to change. Sibling subclasses cannot easily share the same factory method.

A *factory object* is an object that encapsulates factory methods.

```
class BicycleFactory {
    Frame createFrame() { return new Frame(); }
    Wheel createWheel() { return new Wheel(); }
    Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
        return new Bicycle(frame, front, rear);
    }

    // return a complete bicycle without needing any arguments
    Bicycle completeBicycle() {
        Frame frame = createFrame();
        Wheel frontWheel = createWheel();
        Wheel rearWheel = createWheel();
        return createBicycle(frame, frontWheel, rearWheel);
    }
}

class RacingBicycleFactory {
    Frame createFrame() { return new RacingFrame(); }
}
```

```

Wheel createWheel() { return new Wheel700c(); }
Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new RacingBicycle(frame, front, rear);
}
}

class MountainBicycleFactory {
    Frame createFrame() { return new MountainFrame(); }
    Wheel createWheel() { return new Wheel26inch(); }
    Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
        return new RacingBicycle(frame, front, rear);
    }
}

```

The Race methods use the factory objects.

```

class Race {

    BicycleFactory bfactory;

    // constructor
    Race() {
        bfactory = new BicycleFactory();
    }

    Race createRace() {
        Bicycle bike1 = bfactory.completeBicycle();
        Bicycle bike2 = bfactory.completeBicycle();
        ...
    }
}

class TourDeFrance extends Race {
    // constructor
    TourDeFrance() {
        bfactory = new RacingBicycleFactory();
    }
}

class Cyclocross extends Race {
    // constructor
    Cyclocross() {
        bfactory = new MountainBicycleFactory();
    }
}

```

In this version of the code, the type of bicycle is still hard-coded into each variety of race. There is a more flexible method which requires a change to the way that clients call the constructor.

```

class Race {

    BicycleFactory bfactory;

    // constructor
    Race(BicycleFactory bfactory) {
        this.bfactory = bfactory;
    }

    Race createRace() {
        Bicycle bike1 = bfactory.completeBicycle();
        Bicycle bike2 = bfactory.completeBicycle();
        ...
    }

}

class TourDeFrance extends Race {
    // constructor
    TourDeFrance(BicycleFactory bfactory) {
        this.bfactory = bfactory;
    }
}

class Cyclocross extends Race {
    // constructor
    Cyclocross(BicycleFactory bfactory) {
        this.bfactory = bfactory;
    }
}

```

This is the most flexible mechanism of all. Now a client can control both the variety of race and the variety of bicycle used in the race, for instance via a call like

```
new TourDeFrance(new TricycleFactory())
```

One reason that factory methods are required is *the first weakness of Java constructors*: Java constructors always return an object of the specified type. They can never return an object of a subtype, even though that would be type-correct (both according to Java subtyping and according to true behavior subtyping as described in lecture 13).

In fact, `createRace` is itself a factory method.

Problem Set 6 contains a use of a factory method in `DirectionsFinder`, which has no public constructor. That omission may have seemed a bit strange, but now you understand the motivation and the advantages of writing in that style. One disadvantage is that it is more verbose to write `DirectionsFinder.getDirectionsFinder()` than to write `new DirectionsFinder()`.

### 2.1.3 Prototype

The prototype pattern provides another way to construct objects of arbitrary types. Rather than passing in a `BicycleFactory` object, a `Bicycle` object is passed in. Its `clone` method is invoked

to create new bicycles; we are making copies of the given object.

```
class Bicycle {
    Object clone() { ... }
}

class Frame {
    Object clone() { ... }
}

class Wheel {
    Object clone() { ... }
}

class RacingBicycle {
    Object clone() { ... }
}

class RacingFrame {
    Object clone() { ... }
}

class Wheel700c {
    Object clone() { ... }
}

class MountainBicycle {
    Object clone() { ... }
}

class MountainFrame {
    Object clone() { ... }
}

class Wheel26inch {
    Object clone() { ... }
}

class Race {

    Bicycle bproto;

    // constructor
    Race(Bicycle bproto) {
        this.bproto = bproto;
    }

    Race createRace() {
```

```

        Bicycle bike1 = (Bicycle) bproto.clone();
        Bicycle bike2 = (Bicycle) bproto.clone();
        ...
    }

}

class TourDeFrance extends Race {
    // constructor
    TourDeFrance(Bicycle bproto) {
        this.bproto = bproto;
    }
}

class Cyclocross extends Race {
    // constructor
    Cyclocross(Bicycle bproto) {
        this.bproto = bproto;
    }
}

```

Effectively, each object is itself a factory specialized to making objects just like itself. Prototypes are frequently used in dynamically typed languages such as Smalltalk, less frequently used in statically typed languages such as C++ and Java.

There is no free lunch: the code to create objects of particular classes must go somewhere. Factory methods put the code in methods in the client; factory objects put the code in methods in a factory object; and prototypes put the code in `clone` methods.

## 2.2 Sharing

Several other design patterns are related to object creation in that they affect constructors (and require the use of factories) and are related to structure in that they specify patterns of sharing among various objects.

### 2.2.1 Singleton

The singleton pattern guarantees that only one object of a particular class ever exists. You might want to use this for `Gym` in your Gym Manager from recitation, because its methods (such as waiting lists for particular machines) are best suited to management of a single location, not oversight from the national office of a chain. A program that instantiates multiple copies probably has an error, but use of the singleton pattern renders such an error harmless.

```

class Gym {
    private static Gym theGym;
    // constructor
    private Gym() { ... }
    // factory method
    public static getGym() {
        if (theGym == null) {

```

```

        theGym = new Gym();
    }
    return theGym;
}
...
}

```

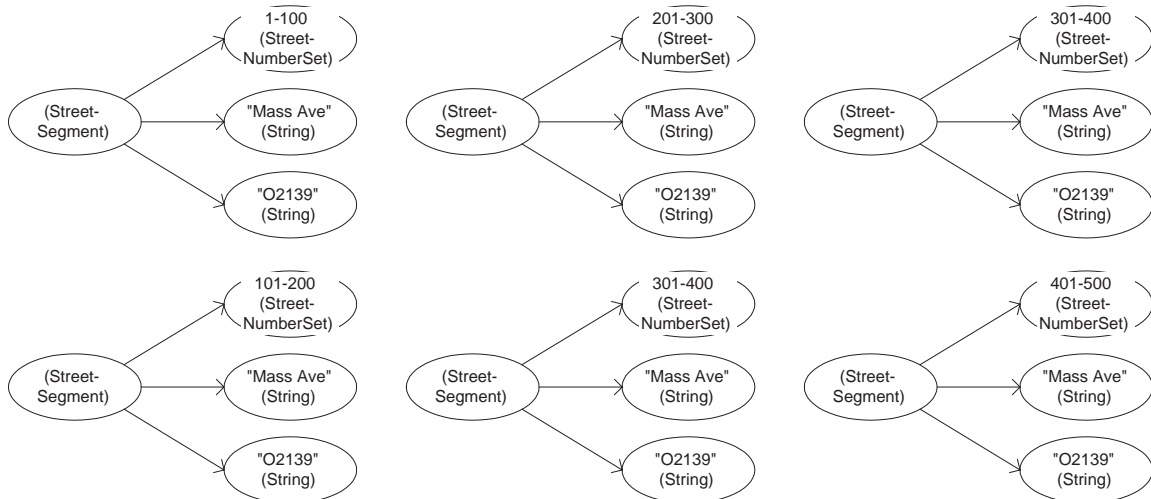
The singleton pattern is also useful for large, expensive objects that should not be multiply instantiated.

The reason that a factory method, rather than a constructor, must be used is *the second weakness of Java constructors*: Java constructors always return a new object, never a pre-existing object.

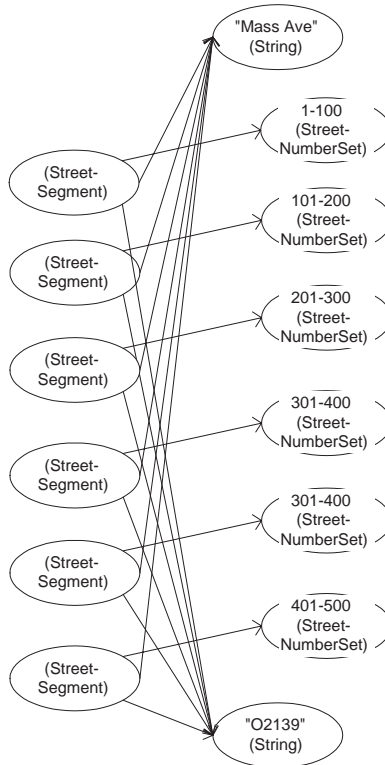
### 2.2.2 Interning

The interning design pattern reuses existing objects rather than creating new ones. If a client requests an object which is equal to one that already exists, then the pre-existing one is returned instead. This is correct only for immutable objects.

As an example, MapQuick may represent a particular street by many **StreetSegment**s. Those **StreetSegment** objects will have the same street name and zipcode. Here is one possible object diagram (snapshot) for a part of the street.



This representation is correct (for instance, all pairs of street names return true when compared with `equals`), but it is unnecessarily wasteful of space. This would be a better runtime configuration of the system:



The difference in space is substantial — enough that it is very unlikely that you could read even a modest database into MapQuick in the absence of this sharing. Therefore, the implementation of `StreetSegReader` which you were provided performs this operation.

Interning arranges for objects that are immutable to be reused — rather than create a new object, a canonical representation is reused. Interning requires a table of all the objects that have ever been created; if it contains any objects that would be equal to the desired object, that version is returned instead. For performance reasons, a hashtable from contents to objects is usually returned (since equality depends only on the contents).

Here is a code fragment that canonicalizes (interns) strings that name segments.

```
HashMap segnames = new HashMap();

canonicalName(String n) {
    if (segnames.containsKey(n)) {
        return segnames.get(n);
    } else {
        segnames.put(n, n);
        return n;
    }
}
```

Strings are a special case, since the best representation for the sequence of characters (the content) is itself a string; we end up with a table from strings to strings. This strategy is correct in general: the code constructs a non-canonical representation, maps it to the canonical representation, and returns the canonical one. However, depending on how much work the constructor is, it may be more efficient not to construct the non-canonical representation if not necessary, in which case

the table might map from contents to canonical representations. In that case, if we were interning `GeoPoints`, we would index the table by the latitude and longitude.

This code uses a map from strings to themselves, but it cannot use a `Set` rather than a `Map`. The reason is that `Sets` do not have a `get` operation, only a `contains` operation. The `contains` operation uses `equals` for its comparison. Thus, even if `myset.contains(mystring)`, that doesn't mean that `mystring` is identically a member of `myset`, and there is no convenient way to access the element of `myset` that `equals mystring`.

The notion of having only one version of a given string is such an important one that it is built into Java; `String.intern` returns the canonical version of a string.

The Liskov text discusses interning in section 15.2.1, but calls the pattern “flyweight,” which is different than the standard terminology in the field.

### 2.2.3 Flyweight

Flyweight is a generalization of interning. (Liskov text section 15.2.1, titled “Flyweight,” discusses interning, which is a special case of flyweight.) Interning is applicable only when an object is completely immutable. The more general form of flyweight can be used when most (but not necessarily all) of an object is immutable.

Consider the case of bicycle spokes.

```
class Wheel {
    ...
    FullSpoke[] spokes;
    ...
}

// We'll name a trimmed-down version "Spoke", so call this "FullSpoke"
class FullSpoke {
    int length;
    int diameter;
    boolean tapered;
    Metal material;
    float weight;
    float threading;
    boolean crimped;
    int location;    // which rim and hub holes this is installed in
}
```

There are typically 32 or 36 spokes per wheel (up to 48 per wheel for a `TandemBicycle`). However, there are usually only three different varieties of spoke per bicycle: one for the front wheel and two for the rear wheel (because the rear hub is off-center, so different lengths are required). We would prefer to allocate just three different `Spoke` (or `FullSpoke`) objects rather than one per spoke of the bicycle. It's not acceptable to have a single `Spoke` object in `Wheel` rather than an array not just because of the asymmetry of the rear wheel but also because I might replace a spoke (say, after one breaks) with another that has the same length but differs in other characteristics. Interning is not an option because the objects are not identical: they differ in their `location` field. In a bicycle rally of 10,000 bicycles, there might only be a few hundred different varieties of spoke but a million instances of them; it would be disastrous to allocate millions of `Spoke` objects. `Spoke`

objects could be shared between different bicycles (two friends with identical bicycles could share the same spoke #22 on the front wheel), but that still leaves a factor of 32 or 36 too little sharing, and in any event it is more likely that there would be similar spokes within a bicycle than across bicycles.

The first step for using the flyweight pattern is to separate the *intrinsic* state from the *extrinsic* state. The intrinsic state is kept in the object; the extrinsic state is kept outside the object. In order to permit interning, the intrinsic state should be both immutable and similar across objects.

Create a location-less `Spoke` class for the intrinsic state:

```
class Spoke {
    int length;
    int diameter;
    boolean tapered;
    Metal material;
    float weight;
    float threading;
    boolean crimped;
}
```

To add the extrinsic state, it does not work to do

```
class InstalledSpokeFull extends Spoke {
    int location;
}
```

because that is just shorthand for `FullSpoke`; `InstalledSpokeFull` takes the same amount of memory as `FullSpoke` because it has the same fields.

Another possibility is

```
class InstalledSpokeWrapper {
    Spoke s;
    int location;
}
```

This is an example of a wrapper (which we will learn more about very soon), and it saves quite a bit of space, because `Spoke` objects can be shared among `InstalledSpokeWrapper` objects. However, there is a solution which uses even less space.

Notice that the location is apparent from the index of the `Spoke` object in the `Wheel.spokes` array:

```
class Wheel {
    ...
    Spoke[] spokes;
    ...
}
```

There is no need to store that (extrinsic) information at all. Some client code (in `Wheel`) must change, because `FullSpoke` methods which used the `location` field must be given access to that information.

Given this version using `FullSpoke`:

```

class FullSpoke {
    // tension the spoke by turning the nipple the specified number of turns
    void tighten(int turns) {
        ... location ...
    }
}

class Wheel {
    FullSpoke[] spokes;

    // The method should be named "true", but that identifier name is a poor choice
    void align() {
        while (wheel is misaligned) {
            ... spokes[i].tighten(numturns) ...
        }
    }
}

```

The corresponding new version using a flyweight spoke is:

```

class Spoke {
    void tighten(int turns, int location) {
        ... location ...
    }
}

class Wheel {
    FullSpoke[] spokes;

    void align() {
        while (wheel is misaligned) {
            ... spokes[i].tighten(numturns, i) ...
        }
    }
}

```

The reference to an interned `Spoke` is so lightweight by comparison with a reference to a non-interned `FullSpoke` that the former is called a flyweight; that could equally apply to `InstalledSpokeWrapper`, though its space overhead is a minimum of three times as great (and possibly more).

The same trick works if `FullSpoke` contains a `wheel` field which refers to the wheel on which it is installed; `Wheel` methods can simply pass `this` to the `Spoke` method.

If `FullSpoke` also contains a boolean field `broken`, how can that be represented? It is also extrinsic information, but it does not appear in the program implicitly as the location and wheel do. It must be stored explicitly in `Wheel`, probably as a `boolean[]` which parallels the `spokes` array. This is slightly unfortunate—the code is starting to get ugly—but acceptable if space savings are critical. If there are many such fields, however, the design should be reconsidered.

Remember that flyweight should only be used at all after profiling has determined that space usage is a critical bottleneck. Introducing such constructs into programs complicates them and presents many opportunities for error. It should be undertaken only in very limited circumstances.

## 3 Structural patterns

### 3.1 Wrappers

Wrappers modify the behavior of another class; they are usually a thin veneer over the encapsulated class, which does the real work. The wrapper may modify the interface, extend the behavior, or restrict access. The wrapper intermediates between two incompatible interfaces, translating calls between the interfaces. This permits two pieces of code that were not designed or written together, and thus are slightly incompatible, to be used together anyway.

Three varieties of wrappers are adapters, decorators, and proxies:

Pattern	Functionality	Interface
Adapter	same	different
Decorator	different	same
Proxy	same	same

The functionality and interfaces compared are those at the inside and outside of the wrapper; that is, a client's view of the wrapped object is compared to a client's view of the wrapper.

The remainder of this section discusses the three varieties of wrapper, then examine tradeoffs between two implementation strategies, subclassing and delegation.

#### 3.1.1 Adapter

Adapters change the interface of a class without changing its basic functionality. For instance, they might permit interoperability between a geometry package that requires angles to be specified in radians and a client that expects to pass angles in degrees. Here are two other examples:

**Example: Rectangle** Suppose that you have written code that works on `Rectangle` objects and calls their `scale` method.

```
interface Rectangle {
    // grow or shrink this by the given factor
    void scale(float factor);

    // other operations
    float area();
    float circumference();
    ...
}

class myClass {

    void myMethod(Rectangle r) {
        ...
        r.scale(2);
        ...
    }
}
```

Suppose there is another class `NonScaleableRectangle` which lacks the `scale` method but does have the other methods of `Rectangle`, as well as additional `setWidth` and `setHeight` methods.

```
class NonScaleableRectangle {
    void setWidth(float width) { ... }
    void setHeight(float height) { ... }
    ...
}
```

You may wish to switch to (or at least permit use of) this variety of rectangle, perhaps because it has desirable features, such as better performance, or perhaps because it is used elsewhere, in a system with which you need to interoperate.

You cannot use `NonScaleableRectangle` directly because of the incompatible interface. However, you can write an adapter which permits its use. There are two ways to do this: subclassing and delegation. The subclassing solution will be familiar:

```
class ScaleableRectangle1 extends NonScaleableRectangle implements Rectangle {
    void scale(float factor) {
        setWidth(factor * getWidth());
        setHeight(factor * getHeight());
    }
}
```

Delegation is a technique for “passing the buck”, forwarding a request so that a different object does the requested work.

```
class ScaleableRectangle2 implements Rectangle {
    NonScaleableRectangle r;
    ScaleableRectangle2(NonScaleableRectangle r) {
        this.r = r;
    }

    void scale(float factor) {
        setWidth(factor * getWidth());
        setHeight(factor * getHeight());
    }

    float area() { return r.area(); }
    float circumference() { return r.circumference(); }
    ...
}
```

**Example: Palette** Suppose that Professor Jackson calls Professor Ernst late at night because someone has discovered a problem with the problem set: it needs to support bicycles that can be repainted (to change their color). The professors split up the work: Professor Jackson will write a `Palette` class with a method that, given a name like “red” or “blue” or “taupe”, returns an array of three RGB values, and Professor Ernst will write code that uses this class. The professors do so, test their work, and go away for the weekend, leaving the the `.class` files for the TAs to integrate. They find that Professor Ernst has written code that depends on

```

interface Palette {
    // returns RGB values
    int[] getColor(String name);
}

```

but Professor Jackson has implemented a class that adheres to

```

interface BritPalette {
    // returns RGB values
    int[] getColour(String name);
}

```

What are the TAs to do? They do not have access to the source, and they do not have time to reimplement and retest. Their solution is to write an adapter for `BritPalette` that changes the operation name. They can implement the adapter either by subclassing or by delegation.

### 3.1.2 Decorator

Whereas an adapter changes an interface without adding new functionality, a decorator extends functionality while maintaining the same interface. Typically, a decorator does not change existing functionality, only adds to it, so that objects of the resulting class behave exactly like the original ones, but also do something extra.

This sounds like subclassing, but not every instance of subclassing is a decoration. First, the implementation of an operation may be completely different or reimplemented in a subclass; that is not usually the case for a decorator, which contains relatively less functionality and reuses the superclass code. Second, subclasses can introduce new operations; wrappers (including decorators) generally do not.

An example of decoration is a `Window` interface (for a window manager) and a `BorderedWindow` interface. The `BorderedWindow` behaves exactly like the `Window`, except that it also draws a border around the outside.

Suppose that `Window` is implemented like this:

```

interface Window {
    // rectangle bounding the window
    Rectangle bounds();
    // draw this on the specified screen
    void draw(Screen s);
    ...
}

class WindowImpl implements Window {
    ...
}

```

The subclassing implementation would look like this:

```

class BorderedWindow1 extends WindowImpl {
    void draw(Screen s) {
        super.draw(s);
        bounds().draw(s);
    }
}

```

```
    }  
}
```

The delegation implementation would look like this:

```
class BorderedWindow2 implements Window {  
    Window innerWindow;  
  
    BorderedWindow2(Window innerWindow) {  
        this.innerWindow = innerWindow;  
    }  
  
    void draw(Screen s) {  
        innerWindow.draw(s);  
        innerWindow.bounds().draw(s);  
    }  
}
```

### 3.1.3 Proxy

A proxy is a wrapper that has the same interface and the same functionality as the class it wraps. This does not sound very useful on the face of it. However, proxies serve an important purpose in controlling access to other objects. This is particularly valuable if those objects must be accessed in a stylized or complicated way.

For example, if an object is on a remote machine, then accessing it requires use of various network facilities. It is easier to create a local proxy that understands the network and performs the necessary operations, then returns the result. This simplifies the client by localizing network-specific code in another location.

As another example, an object may require locking if it can be accessed by multiple clients. The lock represents the right to read and/or update the object; without the lock, concurrent updates could leave the object in an inconsistent state, or reads in the middle of a sequence of updates could observe an inconsistent state. A proxy could take care of locking an object before an operation or sequence of operations, then unlocking it afterward. This is less error-prone than requiring clients to correctly implement the locking protocol.

Another variety of proxy is a security proxy. It might operate correctly if the caller has the correct credentials (such as a valid Kerberos certificate), but throw an error if an unauthorized user attempts to perform operations.

A final example is a proxy for an object that may not yet exist. If creating an object is expensive (because of computation or network latency), then it can be represented by a proxy instead. That proxy could immediately start to create the object in a background task in the hope that it is ready by the time the first operation is invoked, or it could delay creating the object until an operation is invoked. In the former case, the rest of the system can proceed without waiting; in the latter case, the work of creating the object need never be performed if it is never used. In either case, operations are delayed until the object is ready.

An example of a proxy for a non-existent object is Emacs's autoload functionality. For instance, I have a file `util-mde.el` which defines a number of useful functions. However, I don't want to slow down Emacs by loading it every time I start Emacs. Instead, my `.emacs` file contains code like this:

```
(autoload 'looking-back-at "util-mde")
(autoload 'in-buffer "util-mde")
(autoload 'in-window "util-mde")
```

The form `(autoload 'function "file")` is essentially equivalent to (in Scheme syntax; Emacs Lisp uses `defun`)

```
(define function ()
  (load "file") ;; redefine function
  (function)   ;; call the new version
)
```

Emacs autoloads most of its own functionality, from the mail and news readers to the Java editing mode. People who complain that Emacs starts up too slowly often have put indiscriminate `load` forms in their `.emacs` files; that's like using an inefficient implementation, then complaining that the compiler is poor because the resulting program runs slowly.

Proxy capabilities are particularly useful when clients have no knowledge of whether the object they are manipulating has special properties (such as being located on a remote machine, requiring locking or security, or not being loaded). It is best to insulate the client from such concerns and localize them in a proxy wrapper.

### 3.1.4 Subclassing vs. delegation

Subclassing and delegation are two implementation strategies for implementing wrappers. You are already familiar with subclassing; delegation stores an object in a field, then passes messages to the object.

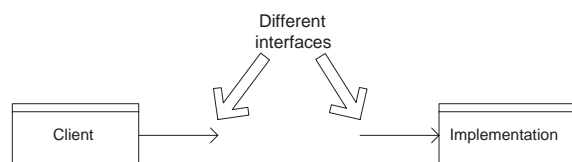
Subclassing automatically gives clients access to all the methods of the superclass. Delegation forces the creation of many small methods like

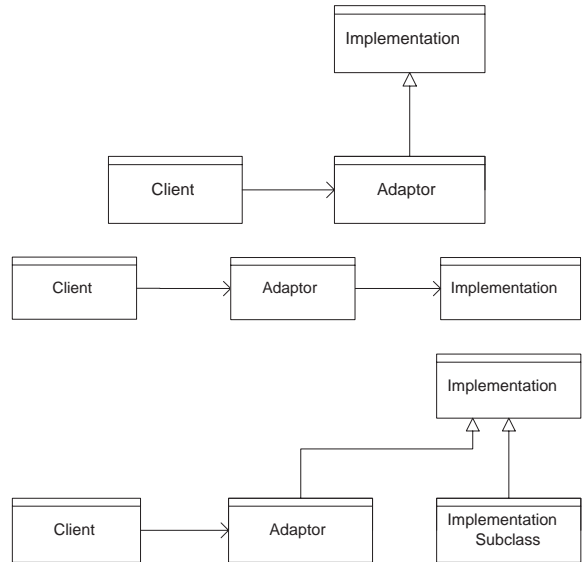
```
void area() { return r.area(); }
```

On the other hand, in order to prevent access to certain methods of the parent, the subclass must override them to throw an error; it would be cleaner not to have them in the interface, which delegation easily permits.

Another potential advantage of subclassing is that it is built into the language; it is likely to be fairly easy to understand and the implementation of subclassing is likely to be fairly efficient.

Delegation is usually the preferred technique for wrappers (and for many design patterns). Wrappers can be added and removed dynamically. For instance, a window can be bordered when active and unbordered otherwise. Another advantage is that objects of arbitrary concrete classes can be wrapped. Creating a subclass specifies the exact type of the object being wrapped. By contrast, the wrapper can wrap an object of any subclass of the declared type of the contained object. Another strength of delegation (related to the previous one) is the ability to use multiple adapters. (For instance, consider how you would create a doubly-bordered window.)





Implementations of the three varieties of wrappers have the same gross structure; without looking at the method bodies to see what work is being done, it is not obvious whether a wrapper is a decorator or a proxy. (An adapter has a different interface than the class it interfaces to.) Some wrappers can even have aspects of more than one variety, though in that case it is clearest for the documentation to say (for instance), “This is both an adapter and a decorator.”

### 3.2 Composite

The composite design pattern permits a client to manipulate either an atomic unit or a collection of units in exactly the same way. The client need not create special code for the case of being given a higher-level object with structure as opposed to being given a basic object; the same operations work on both.

Composite is good for object with part-whole relationships, and the client should not have to worry about whether its argument is atomic or composed of parts.

For example, a bicycle can be decomposed in the following way:

- Bicycle
  - Wheel
    - skewer
    - hub
    - spokes
    - nipples
    - rim
    - tube
    - tire
  - Frame
  - Drivetrain
  - ...

Given a bicycle component, I might want to determine its weight or cost regardless of whether it is itself composed of subcomponents. A client which is given a bicycle component shouldn't have to treat it differently if it is a wheel as opposed to a reflector or a saddle.

The solution to this problem is for all bicycle components to satisfy a common interface:

```
class BicycleComponent {
    int weight();
    float cost();
}
```

The implementation of `Wheel.weight` might itself call `weight` on its subparts, but that is of no import to the client (and the client shouldn't have to worry about that).

An alternative to using a common interface is to have a common superclass; in either way, all bicycle components at all levels provide the same methods and can be used interchangeably.

As another example, a lending library's holdings might be organized into levels as follows:

```
Library
  Section (for a given genre)
    Shelf
      Volume
        Page
          Column
            Word
              Letter
```

If all of these satisfy the interface

```
interface Text {
    String getText();
}
```

then a client can (say) count the number of words, or perform other operations, on as large or as small a part of the library's holdings as desired.

The book gives another example, the syntax of computer programs. Notice that there are two completely unrelated tree structures, illustrated in figures 15.12 and 15.13 on page 392. One is the abstract syntax tree, which breaks down the syntax of a particular utterance in the language, such as a particular block of code. The other is the class hierarchy, which expresses subtyping and inheritance. (In Java, whenever there is inheritance, there is always subtyping, because every subclass is a subtype.) The latter organization permits `Node` to have methods like `typeCheck` or `prettyPrint` which can be called regardless of which concrete `Node` is being operated on. Methods, classes, and packages might also be `Nodes` in this representation.

## 4 Behavioral patterns

### 4.1 Multi-way communication

It is easy enough for a single client to use a single abstraction. (We have seen patterns for easing the task of changing the abstraction being used, which is a common task.) However, occasionally a client may need to use multiple abstractions; furthermore, the client may not know ahead of time how many or even which abstractions will be used. The observer, blackboard, and mediator patterns permit such communication.

### 4.1.1 Observer

Suppose that there is a database of all MIT student grades, and the 6.170 staff wishes to view the grades of 6.170 students. They could write a `SpreadsheetView` class that displays information from the database. (We will assume that the viewer caches information about 6.170 students—it needs this information in order to redraw, for example—but whether it does so is not an important part of this discussion.) The display might look something like this:

	PS1	PS2	PS3
B. Bitdiddle	45	85	80
A. Hacker	95	90	85
A. Turing	90	100	95

Suppose the code to communicate between the grade database and the view of the database uses the following interface:

```
interface GradeDBViewer {
    void update(String course, String name, String assignment, int grade);
}
```

When new grade information is available (say, a new assignment is graded and entered, or an assignment is regraded and the old grade corrected), the grade database must communicate that information to the view. Let's suppose that Ben Bitdiddle has demanded a regrade on problem set 1, and that regrade did reveal grading errors: Ben's score should have been 30. The database code must somewhere make calls to `SpreadsheetView.update`. Suppose that it does so in the following way:

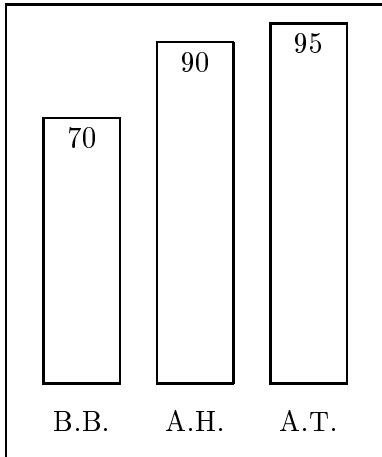
```
SpreadsheetView ssv = new SpreadsheetView();
...
ssv.update("6.170", "B. Bitdiddle", "PS1", 30);
```

(For brevity, this code shows literal values rather than variables for the `update` arguments.)

Then the spreadsheet view would redisplay itself in the following way:

	PS1	PS2	PS3
B. Bitdiddle	30	85	80
A. Hacker	95	90	85
A. Turing	90	100	95

The staff might later decide that they would like to also view grade averages as a bargraph, and implement such a viewer:



Maintaining such a view in addition to the spreadsheet view requires modifying the database code:

```

SpreadsheetView ssv = new SpreadsheetView();
BargraphView bgv = new BargraphView();
...
ssv.update("6.170", "B. Bitdiddle", "PS1", 30);
bgv.update("6.170", "B. Bitdiddle", "PS1", 30);

```

Likewise, adding a pie chart view, or removing some view, would require yet more modifications to the database code. Object-oriented programming (not to mention good programming practice) is supposed to provide relief from such hard-coded modifications: code should be reusable without editing and recompiling either the client or the implementation.

The observer pattern achieves the goal in this case. Rather than hard-coding which views to update, the database can maintain a list of observers which should be notified when its state changes.

```

Vector observers = new Vector();
...
for (int i=0; i<observers.size(); i++) {
    GradeDBViewer v = (GradeDBViewer) observers[i];
    v.update("6.170", "B. Bitdiddle", "PS1", 30);
}

```

In order to initialize the vector of observers, the database will provide two additional methods, `register` to add an observer and `remove` to remove an observer.

```

void register(GradeDBViewer observer) {
    observers.add(observer);
}

boolean remove(GradeDBViewer observer) {
    return observers.remove(observer);
}

```

The observer pattern permits client code (which manages the database and the viewers) to select which observers are active, and observers can even be added and removed at run-time.

This discussion has glossed over a number of details. For instance, the client might store all the information of interest to it (which might be all the 6.170 grades, or just the grades for some students, or just the number of updates to the database for a `DatabaseActivityViewer`), duplicating parts of the database, or the client might read the database when needed. A related design decision is whether the database sends all potentially relevant information to the client when an update occurs (this is the *push* structure), or the database simply informs the client, “an update has occurred” (this is the *pull* structure). The pull structure forces the client to request information, which may result in more messages, but overall a smaller amount of data transferred.

### 4.1.2 Blackboard

The blackboard pattern generalizes the observer pattern to permit multiple data sources as well as multiple viewers. It also has the effect of completely decoupling producers and consumers of information.

A blackboard is a repository of messages which is readable and writable by all processes. Whenever an event occurs that might be of interest to another party, the process responsible for or knowledgeable about the event adds to the blackboard an announcement of the event. Other processes can read the blackboard. In the typical case, they will ignore most of its contents, which do not concern them, but they may take action on other events. A process which posts an announcement to the blackboard has no idea whether zero, one, or many other processes are paying attention to its announcements.

Blackboards generally do not enforce a particular structure on their announcements, but a well-understood message format is required so that processes can interoperate. Some blackboards provide filtering services so that clients do not see all announcements, just those of a particular type; other blackboards automatically send announcements to clients which have registered interest (this is a pull structure).

An ordinary bulletin board (either the physical or the electronic kind) is an example of a blackboard system. Another example of a blackboard at MIT is the zephyr messaging service.

The Liskov text calls this pattern “white board” rather than “blackboard.” The former name may be more modern-seeming, but the latter is standard computer science terminology which has been in use for decades and will be more quickly recognized outside 6.170. The first major blackboard system was the Hearsay-II speech recognition system, implemented between 1971 and 1976.

### 4.1.3 Mediator

The mediator pattern is intermediate between observer and blackboard. It decouples information producers and consumers but does not decouple control. Whereas blackboard communication is asynchronous, mediators are synchronous: they do not return control to the producer before passing the information to all consumers.

## 4.2 Traversing composites

This section discusses traversing composites and/or performing other operations on all the subparts of a composite. Our goal is to support many different operations, and to be able to perform them on many different subparts of a composite. Since both the operation to be performed and the type of composite object to be operated upon affect the implementation, deciding how to break down the problem can be difficult.

Consider the example of an *abstract syntax tree*, or AST, which is a representation of (the syntax of) a computer program. For instance, the binary addition operator `+` might be represented by `PlusOp` objects:

```
class PlusOp extends Expression {
    Expression leftExp;
    Expression rightExp;
    ...
}
```

Variable references, assignment operations (`a=b`), and conditional expressions (`a?b:c`) are other types of expression:

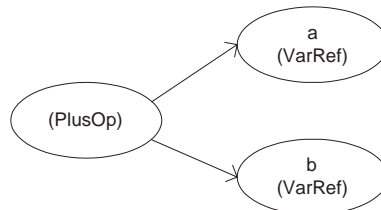
```
class VarRef extends Expression {
    String varname;
    ...
}
```

```
class AssignOp extends Expression {
    VarRef lvalue;           // left-hand side; "a" in "a=b"
    Expression rvalue;      // right-hand side; "b" in "a=b"
    ...
}
```

```
class CondExpr extends Expression {
    Expression condition;
    Expression thenExpr;
    Expression elseExpr;
    ...
}
```

A complete representation would have many other AST node types as well, such as `AssignOp` for assignments, for expressions, and so forth.

A particular use of `+`, such as `a + b`, would be represented at runtime by



A compiler or other program analysis tool creates an AST by parsing the target program; after parsing, the tool performs operations, such as typechecking, pretty-printing, optimizing, or generating code, on the AST. Each operation is different than the others, but each AST node is also unlike the others.

Each box of this table will be filled in with a different piece of code:

		Objects	
		CondExpr	AssignOp
Oper- ations	typecheck		
	pretty-print		

The question is whether to organize the code so as to group all the typechecking code together (and necessarily spread code dealing with `CondExprs` across the implementation) or to group all the code dealing with a particular type of expression, but split up code dealing with a particular operation.

(A related issue is how to select and execute the proper block of code, regardless of where it may be located. Java's method dispatch mechanism selects which version of an overloaded method to call based on the run-time type of the receiver. This makes it possible to dispatch based on either operations or objects, but not both at the same time.)

The interpreter and procedural patterns (and visitor, a refinement of procedural) permit expression of operations over composite objects such as ASTs. Interpreter collects together similar objects and spreads apart similar operations. Procedural collects similar operations and spreads apart similar objects. That means that

Interpreter makes it easy to add objects, hard to add operations.

Procedural makes it easy to add operations, hard to add objects.

“Easy” and “hard” refer to how many different classes need to be modified. When the interpreter class is used, adding a new object requires writing a single new class, but adding a new operation requires modifying every existing class. The reverse is true for procedural. Both patterns have classes for all objects that capture those objects' idiosyncrasies, as illustrated in the code examples for `CondExpr` and `AssignOp` above; the question is where to place the implementations of operations that exist for all objects. The examples below should clarify this notion.

Which approach should you take when designing a software system depends on two factors. First, do you view the system as operation-centric or operand-centric? Are the algorithms central, or are the objects? (In an object-oriented system, often the objects are.) Second, what aspects of the system are most likely to change? (A programming language's syntax rarely changes to add new types of expression, but a program analyzer such as a compiler is often extended with new functionality.) Those changes should be eased by your choice of design pattern.

#### 4.2.1 Interpreter

The interpreter pattern groups together all the operations for a particular variety of object. It uses the pre-existing classes for objects and adds to each class a method for each supported operation. For example,

```
class Expression {
    ...
    Type typecheck();
    String prettyPrint();
}

...

class AssignOp extends Expression {
    ...
    Type typecheck() { ... }
    String prettyPrint() { ... }
}
```

```

class CondExpr extends Expression {
  ...
  Type typecheck() { ... }
  String prettyPrint() { ... }
}

```

#### 4.2.2 Procedural

The procedural pattern groups together all the code that implements a particular operation. It creates a class for each operation; the class has a separate method for each type of operand. For example, typechecking code might look like this:

```

class Typecheck {
  ...
  // typecheck "a?b:c"
  Type tcCondExpr(CondExpr e) {
    Type condType = tcExpression(e.condition); // type of "a"
    Type thenType = tcExpression(e.thenExpr); // type of "b"
    Type elseType = tcExpression(e.elseExpr); // type of "c"
    // BoolType is defined elsewhere
    if ((condType == BoolType) && (thenType == elseType)) {
      // This expression is well-typed, because the condition is of boolean
      // type and the then and else branches have the same type.
      // The type of the whole expression is the type of the branches.
      return thenType;
    } else {
      return ErrorType; // ErrorType is defined elsewhere
    }
  }

  // typecheck "a=b"
  Type tcAssignOp(AssignOp e) {
    ...
  }
}

```

The procedural pattern works well enough, but there is one ugly aspect: the definition of `tcExpression`. It needs to call `tcCondExpr` or `tcAssignOp` or `tcVarRef` or some other function, depending on the run-time type of the subcomponents of an expression.

```

class Typecheck {
  ...
  Type tcExpression(Expression e) {
    if (e instanceof PlusOp) {
      return tcPlusOp((PlusOp)e);
    } else if (e instanceof VarRef) {
      return tcVarRef((VarRef)e);
    } else if (e instanceof AssignOp) {

```

```

        return tcAssignOp((AssignOp)e);
    } else if (e instanceof CondExpr) {
        return tcCondExpr((CondExpr)e);
    } else ...
    ...
}
}

```

Maintaining this code is tedious and error-prone, and the long cascaded `if` tests are likely to run slowly. Furthermore, even though this code would be undesirable even if it occurred only once, in fact it occurs again in the `PrettyPrint` class and in every other operation class. Systematic repetition in code is usually a sign for a need to redesign, possibly using a design pattern.

We already know of a Java construct that automatically chooses which code to execute based on a type test: method dispatching. It does the same kind of comparison and selection as the cascaded `if` tests, but does not clutter the code and is likely to be considerably more efficient. The visitor pattern takes advantage of this.

### 4.2.3 Visitor

The visitor pattern encodes a depth-first traversal (or alternately, some other variety of traversal) over a hierarchical data structure such as one resulting from the composite pattern. The visitor pattern depends on two operations: nodes (objects) accept visitors, and visitors visit nodes (objects). Conceptually, the code structure is as follows:

```

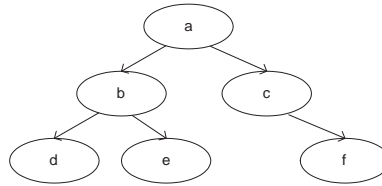
class Node {
    ...
    void accept(Visitor v) {
        for each child of this node {
            child.accept(v);
        }
        v.visit(this);
    }
}

class Visitor {
    ...
    void visit(Node n) {
        perform work on n
    }
}

```

The `accept` and `visit` methods work together so that `n.accept(v)` performs a depth-first traversal of the structure rooted at `n`, with the operation represented by `v` performed on each component of the structure in turn.

Consider a composite with the following structure:



The sequence of calls resulting from `a.accept(v)` for some visitor `v` is:

```

a.accept(v)
  b.accept(v)
    d.accept(v)
      v.visit(d)
    e.accept(v)
      v.visit(e)
    v.visit(b)
  c.accept(v)
    f.accept(v)
      v.visit(f)
    v.visit(c)
  v.visit(a)
  
```

The sequence of calls to `visit`, which performs the actual work, is `d, e, b, f, c, a`; this is a depth-first search. The `visit` method might count the number of nodes, or perform type-checking, or some other operation.

The visitor pattern requires the addition of `visit` and `accept` methods; see the Liskov book for an example. As with the procedural pattern, visitor makes it easy to add operations (visitors) but hard to add nodes (which requires modifying each existing visitor).

A visitor is very much like an iterator: essentially, each element of a data structure is presented in turn to the `visit` method. It gives the opportunity for more, however: a visitor can accumulate state that would be impossible to determine from the sequence of nodes alone. Unfortunately, the implementation structure described above does not provide any way for one call of `visit` to communicate with another.

Here are two possible solutions to this problem. The book proposes saving information in a separate data structure (for example, a stack) which can be read and written. This keeps the visitors and acceptors clean, but it can be hard to see how data flows between calls.

An alternate solution is to move some of the work into the visitor itself:

```

class Node {
  ...
  void accept(Visitor v) {
    v.visit(this);
  }
}

class Visitor {
  ...
  void visit(Node n) {
    for each child of this node {
      child.accept(v);
    }
  }
}
  
```

```
    }  
    perform work on n  
  }  
}
```

This solution has several problems. For one thing, there are many visitors, so the traversal code is repeated many times rather than just appearing once (since there is only one acceptor). Second, the acceptor is not really doing anything any more. The visitor is essentially doing a depth-first search of its own. This solution does have the merit of making the information flow clearer, in the common case that a visitor for a node depends on the results of from visiting children.

### 4.3 State

We will not discuss the state pattern in detail, but you might want to consider it for implementing `StreetNumberSet`.