

CS 4350: Fundamentals of Software Engineering
CS 5500: Foundations of Software Engineering

Lesson 2.4 Design Patterns (Part 2)

Jon Bell, John Boyland, Mitch Wand
Khoury College of Computer Sciences

Learning Objectives for this Lesson

- By the end of this lesson you should be able to:
 - Explain and illustrate the following patterns
 - Singleton
 - Observer
 - Visitor

Review: Definition of a Pattern

- Alexander says:

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

Review: Design Patterns are Controversial

- You can get into endless debates about whether a given piece of code is or is not a correct instance of a particular pattern.
- We are just not going to get into that.
- These patterns are tools in your toolbox.
- But keep your 5004/5010 notes close at hand.

Problem #1: make sure there is only one instance of a particular class

- Primary examples:
 - a clock
 - a storage allocator
 - a generator for unique identifiers

Solution: The Singleton Pattern

- We'll do this in stages

A Simple Clock

```
export interface IClock {  
    // reset the tick counter to 0  
    reset(): void  
    // increment the tick counter  
    tick(): void  
    // returns the number of ticks since the last reset.  
    currentTime(): number  
}  
  
class Clock implements IClock {  
    private ticks = 0  
    public reset():void { this.ticks = 0 }  
    public tick():void { this.ticks++ }  
    public currentTime():number { return this.ticks }  
}
```

A Clock Factory

```
class FactoryMadeClock implements IClock {  
    private ticks = 0  
    public reset():void { this.ticks = 0 }  
    public tick():void { this.ticks++ }  
    public currentTime():number { return this.ticks }  
}  
  
// no need to instantiate ClockFactory  
// just say ClockFactory.getClock()  
export default class ClockFactory {  
    public static getClock():IClock {return new FactoryMadeClock()}  
}
```

calling
`ClockFactory.getClock()`
returns a new clock

Note that `getClock` is static,
so you don't need to
instantiate `ClockFactory`.

This is an instance of the **Factory Pattern** (yet another pattern whose name you should know). This pattern doesn't add much value here, but it would be helpful if you were building something more complicated, e.g. an Amazon product listing.

A Singleton Clock Factory

```
class Clock {  
    ..same as before..  
}  
  
export default class SingletonClockFactory {  
    private constructor() {}  
  
    private static _theClock: IClock  
  
    // have we initialized the clock?  
    private static _isInitialized : boolean = false  
  
    public static getClock() {  
        if (!this._isInitialized) {  
            this._theClock = new Clock()  
            this._isInitialized = true // it's initialized now  
        }  
        return this._theClock  
    }  
}
```

Like the ClockFactory, but this one cheats and only makes a clock once. Then it returns that same clock every time.

Make the factory's constructor private, so that no one can create another one

Use a first-time-through switch

Let's test this...

```
import {assert} from 'chai'
import SingletonClockFactory from './SingletonClockFactory'

function test1 () {
  let clock1 = SingletonClockFactory.getClock()
  let clock2 = SingletonClockFactory.getClock()
  clock1.tick()
  assert.equal(clock1.currentTime(),1)
  clock1.tick()
  assert.equal(clock1.currentTime(),2)
  assert.equal(clock2.currentTime(),2, "clock2 should see clock1's ticks")
  clock2.tick()
  assert.equal(clock1.currentTime(),3, "clock1 should see clock2's ticks")
}

describe ('check that clock is a singleton', () => {
  it('test1', test1)
})
```

Problem #2

- You have an object that changes state, and there are many other objects in the system that need to know this.
- But you don't know who they are—
 - they may even be created after the object that is being watched
- Example: we have a master clock, and other objects need to know the current time.

Solution: The Observer Pattern

- Also called "publish-subscribe"
- The object being observed (the "subject") keeps a list of the people who need to be notified when something changes.
- When a new object wants to be notified when the subject changes, it registers with ("subscribes to") with the subject

Interfaces

```
export interface IPublishingClock {  
  // reset the tick counter  
  reset(): void  
  // increment the tick counter  
  tick(): void  
  // subscribe a new observer  
  subscribe(obs:ClockObserver) : void  
}
```

```
export interface ClockObserver {  
  // action to take when clock ticks  
  onTick(time:number):void  
  
  // action to take when the clock resets  
  onReset():void  
}
```

No 'getTime' method! The clock *pushes* information to the observers

The protocol is:

1. When the clock ticks, it sends an onTick message with the current time to each subscriber (observer)
2. When the clock resets, it sends an onReset message to each subscriber.
3. When a new subscriber registers, the clock responds by sending it an onTick message

Names like 'onTick' are typical for methods in the Observer pattern

The Clock

```
class Clock implements IPublishingClock {  
  
    // clock functionality  
    private clockTime = 0  
    public tick () {this.clockTime++; this.publishTickEvent()}  
    public reset() {this.clockTime=0; this.publishResetEvent()}  
  
    private observers : ClockObserver[]  
  
    // register responds with the current time, so the observer  
    // will be initialized  
    public subscribe(obs:ClockObserver): void {  
        this.observers.push(obs);  
        obs.onTick(this.clockTime)  
    }  
    private publishTickEvent() {  
        this.observers.forEach(obs => {obs.onTick(this.clockTime)})  
    }  
  
    private publishResetEvent() {  
        this.observers.forEach(obs => {obs.onReset()})  
    }  
}
```

Push vs Pull

- In the simple model (like the one in singleton), a client **pulled** information from the clock.
- In the observer model, the clock **pushes** information to its clients

Exercise: Draw UML sequence diagrams for the simple clock and for the publishing clock.

Last Problem

- You have a hierarchical structure, and there are many operations that will need to traverse it.
- You don't know in advance what those operations will be.
- But each operation can be implemented imperatively, perhaps by accumulating the answer in some variable.
- Also, you'd like to keep the internal organization of each node in the structure hidden from the operation.

Solution: The Visitor Pattern

- Represent each operation as a class, with a method for each kind of node you have.
- To invoke the operation, create a new object of the Visitor class.
- Then send the visitor to the node.
- The node calls back the appropriate method of the visitor, and then sends the visitor on to each of its children.

This is called the **Visitor** class.

Let's call that the **visitor** (with a small v).

Let's apply this to the shapes example

```
// operates on a node
// the node itself is responsible for invoking the
// visitor on its descendants, if any.
interface ShapeVisitor {
    visitCircle(c:Circle): void
    visitSquare(sq:Square): void
    visitCompound(c:Compound): void
}

// a Shape is any class that will accept a Shape Visitor
interface Shape {
    // calls back the appropriate method of the visitor.
    // also sends the visitor to each child of the shape
    accept (v:ShapeVisitor) : void
}
```

I think 'accept' is a terrible name for this, but it's what everybody calls it. So if you see a method called 'accept' or 'acceptVisitor' in a codebase, that probably means that there's a visitor pattern here.

A typical shape definition

```
class Compound implements Shape {  
  
    public accept (v:ShapeVisitor) {  
        // apply the visitor using in-order traversal  
        this.back.accept(v);  
        v.visitCompound(this);  
        this.front.accept(v)}  
  
    constructor(private front:Shape  
  
    public getFront () : Shape { re  
    public getBack  () : Shape { re  
  
}
```

It's up to the node to decide the order in which these operations happen. This order is called **in-order** traversal. Other possible orders are called **pre-order**, and **post-order**. You should have learned what those mean back in your data structures class. If you didn't, go look it up. It's important vocabulary for any computer scientist.

When a Compound accepts a visitor, it

1. Passes the visitor on to its back shape.
2. Sends itself to the appropriate method of the visitor for local processing
3. Passes the visitor on to its front shape.

The front and back properties are private to preserve encapsulation. We need getters to make their values available to `v.visitCompound`. Or you could make them public if you wanted to allow the visitor (or anybody else) to change them.

A typical visitor

```
// creates a list of all the ScreenPositions in the shape that
// it visits.
// retrieve the final answer with getPositions()
class ListPositionsVisitor implements ShapeVisitor {
    // a list in which to accumulate the positions as we find them
    private positions : ScreenPosition[] = []

    // for Circle or Square, accumulate their position in the list
    public visitCircle (c:Circle) {this.positions.push(c.getPos())}
    public visitSquare (sq:Square) {this.positions.push(sq.getPos())}

    // a Compound does not have a position, so there's
    // nothing to do here. The node will be responsible for visiting
    // its children. In any case, the accept method does all the work
    public visitCompound (c:Compound) {}

    // report the results
    public getPositions() {return this.positions}
}
```

IMPORTANT: Here the visitCircle method can be sure that its argument is a Circle, not just a Shape. So we can be sure it has a getPos method. If we only knew that c was a Shape, we couldn't be sure that it had such a method.

Exercise: Draw UML sequence diagrams for the invocation of a visitor

Package this up as a function

Applying the
DRY Principle...

```
// given a Shape, returns a list of all the ScreenPosition  
s  
// in the Shape.  
function shape2list(shape: Shape): ScreenPosition[] {  
    let newVisitor = new ListPositionsVisitor()  
    shape.accept(newVisitor)  
    return newVisitor.getPositions()  
}
```

Exercise: Draw UML sequence
diagrams for the invocation
of a visitor

Many variations

- Instead of returning itself, the node could return some of its fields, or hide others from the visitor.
- It's also possible to have visitors that return values, by writing something like

```
interface ShapeVisitor<T> {  
    visitCircle(c:Circle): T  
    visitSquare(sq:Square): T  
    visitCompound(c:Compound): T  
}
```

- but getting the types right requires some care— look at the example code if you want to see how it's done.

Review: Learning Objectives for this Lesson

- You should now be able to explain and illustrate the following patterns:
 - Singleton
 - Observer
 - Visitor

Next steps...

- Come to class armed with questions.
- Next week, we'll zoom out to even larger program structures and talk about software architectures.