CS 4530: Fundamentals of Software Engineering

Module 5: Interaction-Level Design Patterns

Adeel Bhutta and Mitch Wand

Khoury College of Computer Sciences

© 2025 Released under the CC BY-SA license

Learning Goals for this Lesson

- By the end of this lesson, you should be able to
  - Explain how patterns capture common solutions and tradeoffs for recurring problems.
  - Explain and give an example of each of the following:
    - The Data-Pull pattern
    - The Listener pattern
    - The Typed-Emitter pattern
    - The Handler-Passing pattern
    - The Singleton pattern

What is a Pattern?

- A Pattern is a summary of a standard solution (or solutions) to a specific class of problems.
- A pattern should contain
  - A statement of the problem being solved
  - A solution of the problem
  - Alternative solutions
  - A discussion of tradeoffs among the solutions.
- For maximum usefulness, a pattern should have a name.
  - So you can say “here I’m using pattern P” and people will know what you had in mind.

Patterns help communicate intent

- If your code uses a well-known pattern, then the reader has a head start in understanding your code.

Patterns are intended to be flexible

- We will not engage in discussion about whether a particular piece of code is or is not a “correct” instance of a particular pattern.

This week we will talk about the interaction scale

The Structural Scale

- key questions: what are the pieces? how do they fit together to form a coherent whole?

The Interaction Scale

- key questions: how do the pieces interact? how are they related?

The Code Scale

- key question: how can I make the actual code easy to test, understand, and modify?

Design at the Interaction Level corresponds to “OOD Design Patterns”

- Four guys in the 90’s wrote a book that lists a lot of patterns.
- But this is not the be-all and end-all of patterns
- We’ll see patterns at lots of different levels.

The Interaction Scale: Examples

1. The Data-Pull Pattern
2. The Observer or Listener Pattern*
3. The Typed-Emitter Pattern
4. The Handler-Passing Pattern
5. The Singleton Pattern*

*These are “official Design Patterns” that you will see in Design Patterns Books

Information Transfer: Push vs Pull

```
class Producer {
    theData : number
}


class Consumer {
    neededData: number
    doSomeWork () {
        doSomething(this.neededData)
    }
}
```

- How can we get a piece of data from the producer to the consumer?

Pattern 1: consumer asks producer (“data-pull")

```
class Producer {
    theData: number
    getData() { return this.theData }
}

class Consumer {
    constructor(private producer: Producer) { }
    neededData: number
    doSomeWork() {
        this.neededData = this.producer.getData()
        doSomething(this.neededData)
    }
}
```

- The consumer knows about the producer
- The producer has a method that the consumer can call
- The consumer asks the producer for the data

IPullingClock.ts

Example: Interface for a pulling clock

```
export default interface IpullingClock {

    /** sets the time to 0 */
    reset():void

    /** increments the time */
    tick():void

    /** returns the current time */
    getTime():number
}
```

- The interface for a simple clock

simpleClockUsingPull.test.ts

Testing the clock and the client

```
import { SimpleClock, ClockClient } from "./simpleClockUsingPull";

test("test of SimpleClock", () => {
    const clock1 = new SimpleClock
    expect(clock1.getTime()).toBe(0)
    clock1.tick()
    clock1.tick()
    expect(clock1.getTime()).toBe(2)
    clock1.reset()
    expect(clock1.getTime()).toBe(0)
})
```

```
test("test of ClockClient", () => {
    const clock1 = new SimpleClock
    expect(clock1.getTime()).toBe(0)
    const client1 = new ClockClient(clock1)
    expect(clock1.getTime()).toBe(0)
    expect(client1.getTimeFromClock()).toBe(0)
    clock1.tick()
    clock1.tick()
    expect(client1.getTimeFromClock()).toBe(2)
})
```

simpleClockUsingPull.ts

simpleClockUsingPull.ts

```
import IClock from "./IPullingClock";

export class SimpleClock implements IClock {
    private time = 0
    public reset () : void {this.time = 0}
    public tick () : void { this.time++ }
    public getTime(): number { return this.time }
}

export class ClockClient {
    constructor (private theclock:IClock) {}
    getTimeFromClock ():number {
       return this.theclock.getTime()
    }
}
```

SimpleClock is the Producer

ClockClient is the Consumer

But there's a potential problem here.

- What if the clock ticks once per second, but there are dozens of clients, each asking for the time every 10 msec?
- Our clock might be overwhelmed!
- Can we do better for the situation where the clock updates rarely, but the clients need the values often?

Pattern 2: producer tells consumer ("push")

```
class Producer {
    constructor(private consumer: Consumer) { }
    theData: number
    updateData(input) {
        this.theData = doSomethingWithInput(input)
        // notify the consumer about the change:
        this.consumer.notify(this.theData)
    }
}

class Consumer {
    neededData: number
    notify(dataValue: number) {
      this.neededData = dataValue
    }
    doSomeWork() {
        doSomething(this.neededData)
    }
}
```

- Producer notifies the consumer whenever the data is updated
- Probably there will be more than one consumer

This is called the Observer Pattern

- Also called "publish-subscribe pattern"
- Also called "listener pattern"
- The object being observed (the "subject") keeps a list of the objects who need to be notified when something changes.
  - subject = producer = publisher
- When a new object wants to be notified when the subject changes, it registers with ("subscribes to") with the subject/producer/publisher
  - observer = consumer = subscriber = listener

IPushingClockAndClients.ts

Interface for a clock using the Push pattern

```
export interface IPushingClock {

    /** resets the time to 0 */
    reset():void

    /**
     * increments the time and sends a .nofify message with the
     * current time to all the consumers
     */
    tick():void

    /** adds another consumer and initializes it with the current time */
    addListener(listener:IPushingClockClient):number
}
```

IPushingClockAndClients.ts

Interface for a clock listener

```
interface IPushingClockClient {
    /**
     *  * @param t - the current time, as reported by the clock
     */
    notify(t:number):void
}
```

Tests

PushingClock.test.ts

```
test("single observer", () => {
        const clock1 = new PushingClock()
        const observer1
          = new PushingClockClient(clock1)
        expect(observer1.getTime()).toBe(0)
        clock1.tick()
        clock1.tick()
        expect(observer1.getTime()).toBe(2)
    })
```

```
test("Multiple Observers", () => {
    const clock1 = new PushingClock()
    const observer1
      = new PushingClockClient(clock1)
    const observer2
      = new PushingClockClient(clock1)
    const observer3
      = new PushingClockClient(clock1)
    clock1.tick()
    clock1.tick()
    expect(observer1.getTime()).toBe(2)
    expect(observer2.getTime()).toBe(2)
    expect(observer3.getTime()).toBe(2)
})
```

IPushingClockAndClients.ts

A PushingClock class

```
export class PushingClock implements IPushingClock {
    private observers: IPushingClockClient[] = []
    public addListener(obs:IPushingClockClient): number {
        this.observers.push(obs);
        return this.time
    }
    private notifyAll() : void {
            this.observers.forEach(obs => obs.notify(this.time))
        }
    private time = 0
    reset() : void { this.time = 0; this.notifyAll() }
    tick() : void { this.time++; this.notifyAll() }

}
```

IPushingClockAndClients.ts

A Client

```
export class PushingClockClient implements IPushingClockClient
{
    private time:number
    constructor (theclock:IPushingClock) {
        this.time = theclock.addListener(this)
    }

    notify (t:number) : void {this.time = t}
    getTime () : number {return this.time}
}
```

IPushingClockAndClients.ts

Interface for a clock listener

```
interface IPushingClockClient {
    /**
     *  * @param t - the current time, as reported by the clock
     */
    notify(t:number):void
}
```

We could have called this **onTick**

The observer gets to decide what to do with the notification

```
export class DifferentClockClient implements IPushingClockClient {

    /** TWICE the current time, as reported by the clock */
    private twiceTime:number

    constructor (theclock:IPushingClock) {
        this.twiceTime = theclock.addListener(this) * 2
    }

    /** list of all the notifications received */
    public readonly notifications : number[] = [] // just for fun

    notify(t: number) : void {
        this.notifications.push(t)
        this.twiceTime =  t * 2 }

    getTime() : number { return (this.twiceTime / 2) }
}
```

Better test this, too

```
test("test of DifferentClockClient", () => {
        const clock1 = new PushingClock()
        const observer1 = new DifferentClockClient(clock1)
        expect(observer1.getTime()).toBe(0)
        clock1.tick()
        expect(observer1.getTime()).toBe(1)
        clock1.tick()
        expect(observer1.getTime()).toBe(2)
    })
```

Tests for .notifications method

```
test("DifferentClockClient accumulates the times correctly", ()
=> {
        const clock1 = new PushingClock()
        clock1.tick()
        const differentClient = new DifferentClockClient(clock1)
        expect(differentClient.getTime()).toBe(1)
        expect(differentClient.notifications).toEqual([])
        clock1.tick()
        clock1.tick()
        clock1.tick()
        expect(differentClient.getTime()).toBe(4)
        expect(differentClient.notifications).toEqual([2, 3, 4])
    })
```

Push vs. Pull: Tradeoffs

PULL	PUSH
The Consumer knows about the Producer	Producer knows about the Consumer(s)
The Producer must have a method that the Consumer can call	The Consumer must have a method that producer can use to notify it
The Consumer asks the Producer for the data	Producer notifies the Consumer whenever the data is updated
Better when updates are more frequent than requests	Better when updates are rarer than requests

Details and Variations

- How does the consumer get an initial value?
  - Here we've had the producer supply it when the consumer registers
- Should there be an unsubscribe method?
- What data should be passed with the **notify** message?
- How does the producer store its registered consumers?
  - If many consumers, this could be an issue
- "There's a package for that"

Pattern #3: The Typed Emitter Pattern

- What if the data source wants to notify its listeners with several different kinds of messages?
- Maybe with different data payloads?
- And what if we want to take advantage of type-checking?

If the data source needs to push different kinds of values, then typed emitters may be useful

```
import { EventEmitter } from "events"
import TypedEmitter from "typed-emitter"

type ClockEvents = {
    reset: () => void
    tick: (time: number) => void, // carries the current time
}
```

IEmittingClockAndClients.ts

Using an emitter

```
class SampleEmitterServer {
    private emitter = new EventEmitter as TypedEmitter<ClockEvents>
    public getEmitter():TypedEmitter<ClockEvents> {return this.emitter}
    public demo() {
        this.emitter.emit('tick', 1); this.emitter.emit('reset')
    }
}

class SampleEmitterClient {
    constructor (server:SampleEmitterServer) {
        const emitter = server.getEmitter()
        emitter.on('tick', (t:number) => {console.log(t)})
        emitter.on('reset', () => {console.log('reset')})
    }
}
```

Interface for a clock using an emitter

```
export interface IEmittingClock {

    /** resets the time to 0 */
    reset():void

    /**
     * increments the time and sends a .nofify message with the
     * current time to all the consumers
     */
    tick():void

    /** adds another listener; returns the clock's emitter */
    addListener(): TypedEmitter<ClockEvents>
}
```

EmittingClock

```
export class EmittingClock implements IEmittingClock {

    private time = 0

    private emitter = new EventEmitter as TypedEmitter<ClockEvents>

    reset(): void { this.time = 0; this.emitter.emit('reset') }

    tick(): void { this.time++; this.emitter.emit('tick', this.time) }

    public addListener(): TypedEmitter<ClockEvents> { return this.emitter }

}
```

EmittingClockClient

```
export class EmittingClockClient {
    private time = 0 // time is not accurate until the next tick
    constructor(theclock: IEmittingClock) {
        const clock: TypedEmitter<ClockEvents> = theclock.addListener()
        // set up event listeners
        clock.on('tick', (t: number) => { this.time = t })
        clock.on('reset', () => { this.time = 0 })
    }

    getTime(): number { return this.time }
}
```

Pattern #4: The handler-passing pattern

- Let's say we have a server with multiple clients
- And the server needs to summarize actions taken by all its clients.
- And you are not allowed to add new methods to the server.
- Solution: when the server constructs the client, it passes the client a *function* to call.
- We call this function the *handler* for the client's action.
- This pattern is used all the time in REACT.

Expected behavior

```
it('works', () => {

    const server = new Server()
    const client1    = server.newClient()
    const client2    = server.newClient()

    expect(server.nPushes).toBe(0)
    client1.buttonPush()
    expect(server.nPushes).toBe(1)
    client2.buttonPush()
    expect(server.nPushes).toBe(2)
    client1.buttonPush()
    expect(server.nPushes).toBe(3)
})
```

- Here the server constructs two clients, and counts the total number of times either of them calls `buttonPush()`

The Code

```
export class Server {

    /** the total number of times the button
     * (in any of the clients) has been pushed
    */
    private _nPushes = 0

    public get nPushes(): number {
       return this._nPushes
    }

    public newClient(): Client {
        return new Client(() => this._nPushes++)
    }
}
```

The handler for buttonPushes

```
export class Client {

  private handlePush: () => void

  constructor (_handlePush: () => void) {
    this.handlePush = _handlePush
    }

   public buttonPush () {
     this.handlePush()
    }
}
```

Pattern #5: The Singleton Pattern

- Maybe you only want one clock in your system.
- You can't just say "new Clock" because that always creates a new object of class Clock.
- We'll solve this in two steps.

Introduce a clock factory

```
function testClock(clock: IClock, clockName: string) {
    clock.reset()
    clock.tick()
    expect(clock.getTime()).toBe(1)
    clock.tick()
    expect(clock.getTime()).toBe(2)
    clock.reset()
    expect(clock.getTime()).toBe(0)
}

describe('the clock factory should build some working clocks', () => {
    it('works', () => {
        const clock1 = SimpleClockFactory.createClock()
        testClock(clock1, 'clock1')
        const clock2 = SimpleClockFactory.createClock()
        testClock(clock2, 'clock2')
    })
})
```

But we said we wanted only one clock!

- No problem!
- Just modify the factory so it only creates a clock once, and after that just returns the same one over and over again.

singletonClockFactory.test.ts

Here’s the behavior we expect

```
import ClockFactory from './singletonClockFactory'


test("actions on clock1 should be visible on clock2", () => {
    const clock1 = ClockFactory.instance()
    const clock2 = ClockFactory.instance()
    expect(clock1.getTime()).toBe(0)
    expect(clock2.getTime()).toBe(0)
    clock1.tick()
    clock1.tick()
    expect(clock1.getTime()).toBe(2)
    expect(clock2.getTime()).toBe(2)
    clock1.reset()
    expect(clock1.getTime()).toBe(0)
    expect(clock2.getTime()).toBe(0)

})
```

Solution: Use a first-time through switch and a private constructor

singletonClockFactory.ts

```
import IClock from './IPullingClock'
import { SimpleClock } from './simpleClockUsingPull';

export default class SingletonClockFactory  {
    private static theClock : IClock | undefined
    private constructor () {SingletonClockFactory.theClock = undefined}

    public static instance () : IClock {
        if (SingletonClockFactory.theClock === undefined) {
            SingletonClockFactory.theClock  = new SimpleClock
        }
        return SingletonClockFactory.theClock
    }
}
```

Describing your design using these vocabulary words

When I create an object that needs a clock, I ask the master clock factory to issue me a clock, and then I have my new object register itself with the clock.

The master clock updates my object whenever the master clock changes.

The master clock also sends my object an update message when it registers, so my object will always have the latest time.

Discussing your design

Why did you choose this design?

I have a lot of objects, and they each check the time very often. If they were constantly sending messages to the master clock, that would be a big load for it. I sat down with Pat, who is building the master clock, and we agreed on this design.

Discussing your design (2)

How do you know that all of your objects will get the right time?

Pat told me that the master clock is a singleton, so they will all be getting the same time.

The Discussion (3)

Who is responsible for keeping the master clock up to date?

That's something that happens in the module that exports the master clock. Pat is building that module. Pat says it's not hard, but they will show me how to do it in a couple of weeks.

The Discussion (4)

What's to prevent you from ticking the master clock yourself?

The clock factory exports a class with an interface that only allows me to register. The interface doesn’t provide me with a method for ticking the clock.

Learning Goals for this Lesson

- Now that we have come to the end of this lesson, you should be able to
  - Explain how patterns capture common solutions and tradeoffs for recurring problems.
  - Explain and give an example of each of the following:
    - The Data-Pull pattern
    - The Listener pattern
    - The Typed-Emitter pattern
    - The Handler-Passing pattern
    - The Singleton pattern