CS 4350: Fundamentals of Software Engineering
Lesson 2.3 The Interaction Scale

Jonathan Bell, Adeel Bhutta, Ferdinand Vesely, Mitch Wand

Khoury College of Computer Sciences

Learning Goals for this Lesson

- By the end of this lesson, you should be able to
  - Give 4 examples of interaction patterns and describe their distinguishing characteristics
  - Draw a picture or give an example to illustrate each one

The Interaction Scale: Examples

1. The Pull pattern
2. The Push pattern (*The Observer Pattern)
3. *The Factory Pattern
4. *The Singleton Pattern (the lying factory)

*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 ("pull")

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

class Consumer {
    constructor(private src: Producer) { }
    neededData: number
    doSomeWork() {
        this.neededData = this.src.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

Pattern 2: producer tells consumer ("push")

```
class Producer {
    constructor(private target : consumer) {}
    theData : number
    updateData (input) {
        // ..something that changes theData..
        // notify the consumer about the change:
        this.target.notify(this.theData)
    }
}

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

- Producer knows the identity of the consumer
- The Consumer has a method that producer can use to notify it.
- 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

Example: A Clock: AbsClock.ts

```
export default interface AbsClock {

    // 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.ts

```
import AbsClock from "./AbsClock";

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

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

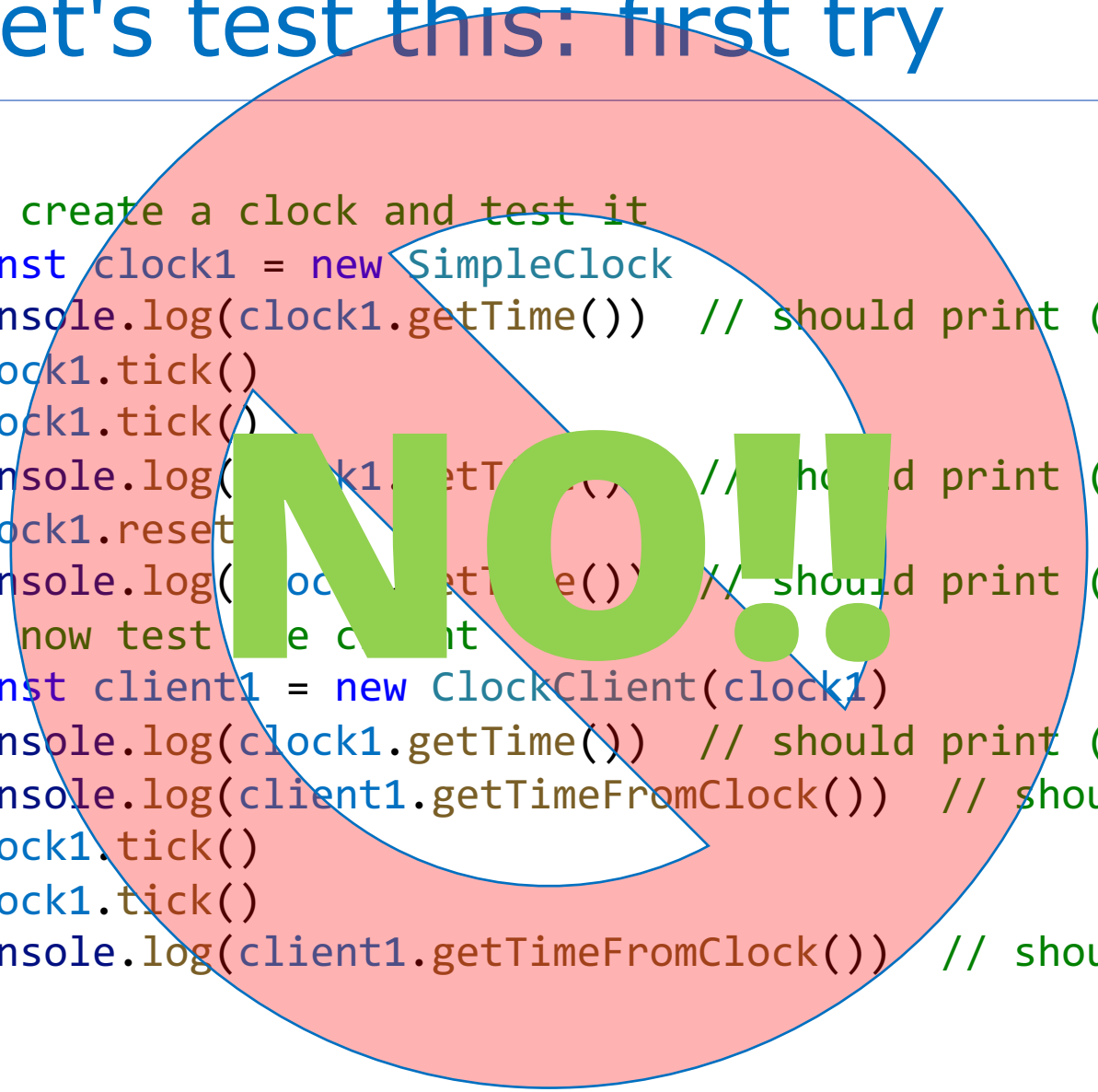
The Producer

The Consumer

Let's test this: first try

index.ts

```
// create a clock and test it
const clock1 = new SimpleClock
console.log(clock1.getTime())  // should print (0)
clock1.tick()
clock1.tick()
console.log(clock1.getTime())  // should print (2)
clock1.reset()
console.log(clock1.getTime())  // should print (0)
// now test the client
const client1 = new ClockClient(clock1)
console.log(clock1.getTime())  // should print (0)
console.log(client1.getTimeFromClock())  // should print (0)
clock1.tick()
clock1.tick()
console.log(client1.getTimeFromClock())  // should print (2)
```



Use automated tests instead

SimpleClockWithPull.spec.ts

```
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)
})
```

Pattern 2: producer tells consumer ("push")

```
class Producer {
    constructor(private target : consumer) {}
    theData : number
    updateData (input) {
        // ..something that changes theData..
        // notify the consumer about the change:
        this.target.notify(this.theData)
    }
}

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

- Producer knows the identity of the consumer
- The Consumer has a method that producer can use to notify it.
- Producer notifies the consumer whenever the data is updated
- Probably there will be more than one consumer

clockUsingPush.ts

Example: ClockUsingPush

```
export interface AbsObservedClock {
    reset():void                                  // resets the time to 0
    tick():void                                   // increments the time and
                                                  // notifies the consumers
    addConsumer(obs:AbsConsumer):void             // adds another consumer
                                                  // to be notified
}
export interface AbsClockConsumer {
    // accepts notification that the current time is t
    notify(t:number):void
}
```

The Clock

```
export class ObservedClock implements AbsObservedClock {
    private observers: AbsClockConsumer[] = []
    public addObserver(obs:AbsConsumer){
            this.observers.push(obs)}
    private notifyAll() {
            this.observers.forEach(obs => obs.notify(this.time))
        }

    time: number = 0
    reset() { this.time = 0 }
    tick() { this.time++; this.notifyAll() }

}
```

A Client

```
export class ObservedClockClient implements AbsClockConsumer {
    constructor (private theclock:AbsObservedClock) {
        theclock.addObserver(this)
    }
    private time = 0   // is this the best way to initialize
                       // the time?
    notify (t:number) {this.time = t}
    getTime () {return this.time}
}
```

Tests

clockUsingPush.spec.ts

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

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

The observer gets to decide what to do with the notification

```
export class DifferentClockClient implements AbsClockConsumer {
    constructor (private theclock:AbsObservedClock) {
        theclock.addObserver(this)
    }
    private time = 0
    private notifications : number[] = [] // just for fun
    notify(t: number) {
        this.time =  t * 2
        this.notifications.push(t)
       }
    getTime() { return (this.time / 2) }
}
```

Better test this, too

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

Details and Variations

- How does the producer get an initial value?
- How does the consumer get an initial value from the producer?
  - maybe it gets it when it subscribes?
  - maybe it should pull it from the producer?
- Should there be an unsubscribe method?

Pattern 3: The Factory Pattern

- The situation:
  - Your task is to write some code that depends only an interface, not on a class that implements it.
  - But your task requires you to create some objects that satisfy the interface.
  - What to do? You can't call 'new', because that would require you to know the class name.
- How to organize this?
  - Create a Factory whose job it is to create the objects.
  - Call the factory when you need a new object.
  - Your code will depend only on the interface, because that's all you have to work with.
- Often our assignments will be structured in this way.
- This is a little confusing; let's look at an example

The Interfaces

```
// from AbsClock.ts, as before...
export default interface AbsClock {
    reset():void
    tick():void
    getTime():number
}

interface AbsClockFactory {
    // returns an object satisfying the AbsClock interface
    instance() : AbsClock
    // returns a string specifying which clock
    // this factory makes
    clockType : string
    // returns the number of clocks created by this factory
    numCreated() : number
}
```

clockFactories.ts

Some Factories...

clockFactories.ts

```
import * as Clocks from './clocks'

class ClockFactory1 implements AbsClockFactory {
    clockType = “Larry"
    numcreated = 0
    public instance() : AbsClock {
        this.numcreated++;
        return new Clocks.Clock1}
    public numCreated() {return this.numcreated}
}

class ClockFactory2 implements AbsClockFactory {
    clockType = “Curly"
    numcreated = 0
    public instance() : AbsClock {
        this.numcreated++;
        return new Clocks.Clock2}
    public numCreated() {return this.numcreated}
}
```

Choose which factory to export

clockFactories.ts

```
// choose which of the factories to export,
// but don't tell anybody which one it is.

export default ClockFactory1
// export default ClockFactory2
// export default ClockFactory3
```

TypeScript has a neat way of doing this.

Test to see that the clock factory produces a working clock

```
import ClockFactory from './clockFactories'

test("test of the Clock produced by the ClockFactory", () => {
    const factory1 = new ClockFactory
    const clock1 = factory1.instance()
    expect(clock1.getTime()).toBe(0)
    clock1.tick()
    clock1.tick()
    expect(clock1.getTime()).toBe(2)
    clock1.reset()
    expect(clock1.getTime()).toBe(0)
})
```

Pattern #4: The Singleton Pattern

- Maybe you only want one clock in your system.
- The factory needn't return a fresh clock every time.
- Just have it return the same clock over and over again.

The Lying Factory

```
import AbsClock from './AbsClock’

// use whichever clock factory is exported from clockFactories
import ClockFactory from './clockFactories'

class SingletonClockFactory  {
    private constructor() {}
    private static isInitialized : boolean = false
    private static theClock : AbsClock
    public static instance () : AbsClock {
        if (!(SingletonClockFactory.isInitialized)) {
            SingletonClockFactory.theClock = (new ClockFactory).instance()
            SingletonClockFactory.isInitialized = true
        };
        return SingletonClockFactory.theClock
    }
}

export default SingletonClockFactory
```

Test to see that only one clock is created

```
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)

})
```

Learning Goals for this Lesson

- At this point, you should be able to
  - Give 4 examples of interaction patterns and describe their distinguishing characteristics
  - Draw a picture or give an example to illustrate each one