# CS 4530: Fundamentals of Software Engineering

# Module 4: Interaction-Level Design Patterns

Adeel Bhutta, Jan Vitek, Mitch Wand

Khoury College of Computer Sciences

# 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.
  - Give 3 examples of interaction patterns and describe their distinguishing characteristics
  - Draw a picture or give an example to illustrate each one

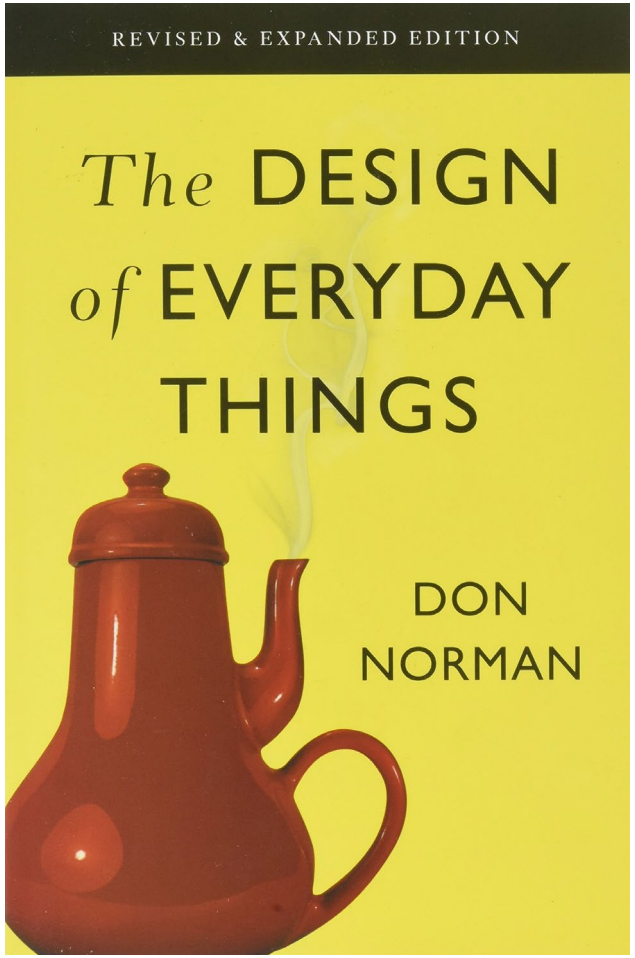
# What is a Pattern?

- 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 make code more comprehensible




# 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.

# Patterns at the Interaction Level correspond 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 Pull pattern
2. The Push pattern (aka the Observer* Pattern or Listener Pattern)
3. 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 ("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

# 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 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

In this lecture, we'll try to stick to Producer/Consumer.

The covey.town project uses 'Listener'

# 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 |

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.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

# Testing the clock and the client

simpleClockUsingPull.test.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 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 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

IPushingClockAndClients.ts

# Interface for a clock using the Push pattern

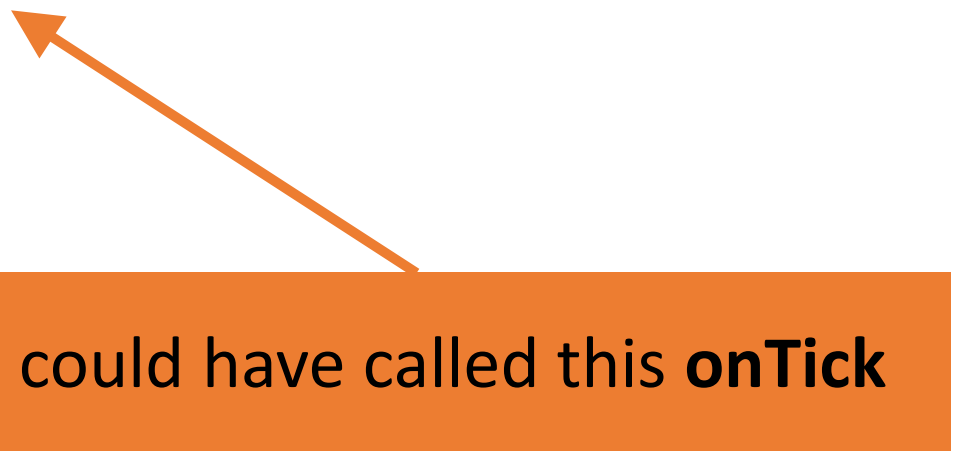
```
export interface IPushingClock {

    /** 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 */
    addListener(listener:IPushingClockClient):void
}
```

# Interface for a client

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

# Interface for a clock listener

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



# A PushingClock class

```
export class PushingClock implements IPushingClock {
    time: number = 0
    reset() { this.time = 0 }
    tick() { this.time++; this.notifyAll() }

    private observers: IPushingClockClient[] = []
    public addConsumer(obs:IPushingClockClient) {
      this.observers.push(obs)
    }
    private notifyAll() {
      this.observers.forEach(obs => obs.notify(this.time))
        }
}
```

# A Client

```
export class PushingClockClient implements IPushingClockClient
{
    constructor (private theclock:IPushingClock) {
        theclock.addConsumer(this)
    }
    // is this the best way to initialize the time?
    private time = 0

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

# Discussion

- Is initializing time to 0 the best way to initialize the client's time?
- How could we better arrange to initialize the clock client?

# 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)
})
```

# The observer gets to decide what to do with the notification

```
export class DifferentClockClient implements IPushingClockClient {
    constructor (private theclock:IPushingClock) {
        theclock.addObserver(this)
    }
    private twicetime = 0  // twice the last time we received
    private notifications : number[] = [] // just for fun
    notify(t: number) {
        this.twicetime =  t * 2
        this.notifications.push(t)
       }
    getTime() { 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)
    })
```

# 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 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.

# Here’s the behavior we expect

singletonClockFactory.test.ts

```
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 s and a private constructor

singletonClockFactory.ts

```
import IClock from './IPullingClock'

class Clock1 implements IClock {
    // implementation of IClock
}

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 Clock1
        }
        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 clock factory. Pat is building that module. They say 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

- By the end of this lesson, you should be able to
  - Explain how patterns capture common solutions and tradeoffs for recurring problems.
  - Give 3 examples of interaction patterns and describe their distinguishing characteristics
  - Draw a picture or give an example to illustrate each one