# CS 4530: Fundamentals of Software Engineering

# Module 4: Interaction-Level Design Patterns

Jonathan Bell, Adeel Bhutta, 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 4 examples of interaction patterns and describe their distinguishing characteristics
  - Draw a picture or give an example to illustrate each one

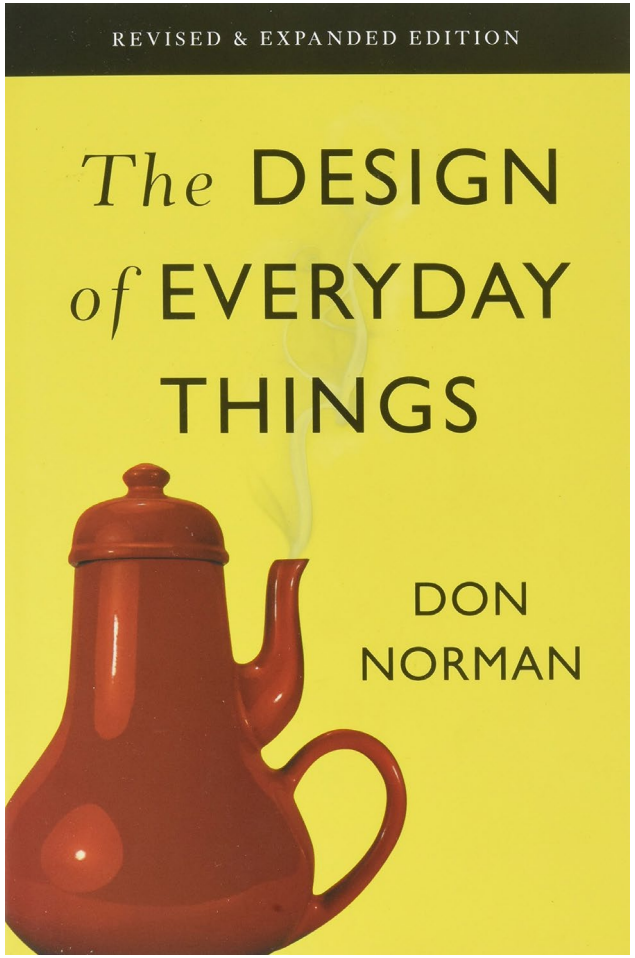
# What is a Pattern?

- A pattern 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 Factory* Pattern
4. The Singleton Pattern* (aka 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

# 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 |

# Example: A Clock: IClock.ts

```
export default interface IClock {

    // 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 IClock from "./IClock";

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()
    }
}
```

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     etTi     )  /     uld pri     )
clock1.rese
console.log     ck1     Ti     )  /     uld pri     )
// now test     cli
const clier     new          ent(c
console.log     ck1.ge          )  // s          (0)
console.log(client1.getTimeFromClock())  // should print (0)
clock1.tick()
clock1.tick()
console.log(client1.getTimeFromClock())  // should print (2)
```

NO!!

# Use automated tests instead

simpleClockWithPull.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 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

# Interface for a clock using the Push pattern

clockUsingPush.ts

```
export interface IProducerClock {

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

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

    // adds another consumer
    addConsumer(listener:IClockConsumer):void
}
```

# Interface for a clock listener

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

# Review: TypeScript interfaces

```
// getx(), gety() return the x,y coordinates of the point
interface IPoint {getx():number, gety():number}

class CartesianPoint implements IPoint {
    constructor (private x : number, private y : number) {}
    getx() {return this.x}
    gety() {return this.y}
}

// r is radius, theta is angle (in radians)
class PolarPoint implements IPoint {
    constructor (private r:number, private theta:number) {}
    getx() {return this.r * Math.cos(this.theta)}
    gety() {return this.r * Math.sin(this.theta)}
}

const point1 = new CartesianPoint(0.0, 1.0)
const point2 = new PolarPoint(1.0, Math.PI/2.0)
```

Go review your Typescript materials if you need to and then come back to this lesson...

# Interfaces are where we specify behaviors

- A temperature sensor is something that returns the current temperature at the sensor's location:

```
// temperatures are measured in Celsius
type Temperature = number

interface AbsTemperatureSensor {
    // returns the current temperature at the sensor location
    getTemperature () : Temperature
}
```

- Note that the interface specifies both syntax (the method name) and the semantics (what the method returns or what it does).

# OO Principle 1: Make Your Interfaces Meaningful

- Interfaces are the thing we use to specify the behavior of the classes and objects that implement them.
- We use the word *behavior* to mean what a single method does:
  - Returning a value is a behavior
  - Having some kind of side-effect (mutation, I/O, etc.) is a behavior

# But the compiler only checks syntax, not semantics

- If we defined a class that had a getTemperature method, but that did not return the temperature at the sensor location, this would not be a correct implementation of AbsTemperatureSensor. For example:

```
class NotReallyASensor implements AbsTemperatureSensor {
    getTemperature () {return 42}
}
```

- The compiler would accept this, but we shouldn't.

Just for fun, make up 3 more classes that the compiler would accept but are not correct implementations of AbsTemperatureSenso r.

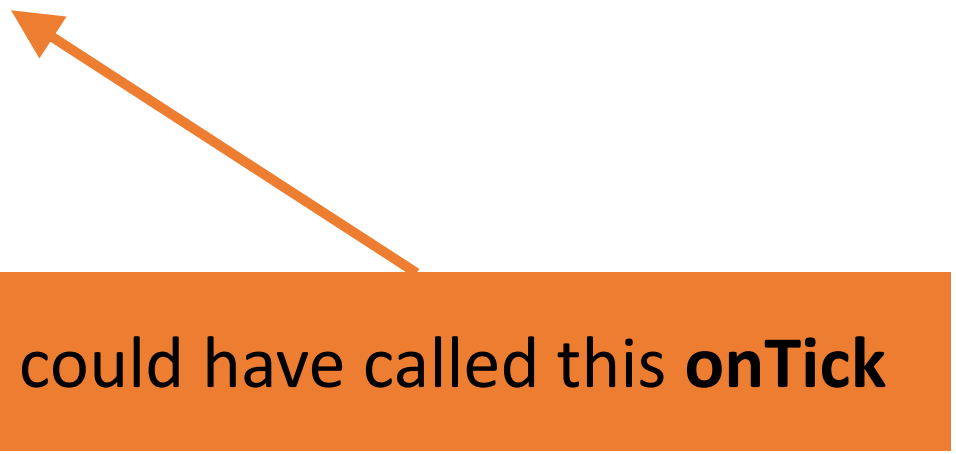
# Interface for a clock using the Push pattern

clockUsingPush.ts

```
export interface IProducerClock {

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

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

    // adds another consumer
    addConsumer(listener:IClockConsumer):void
}
```

# Interface for a clock listener

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



# A ProducerClock class

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

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

# A Client

```
export class ObservedClockClient implements IClockConsumer {
    constructor (private theclock:IProducerClock) {
        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

clockUsingPush.test.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 IClockConsumer {
    constructor (private theclock:IProducerClock) {
        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 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 IClock.ts, as before...
export default interface IClock {
    reset():void
    tick():void
    getTime():number
}

interface IClockFactory {
    // returns an object satisfying the IClock interface
    instance() : IClock
    // 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 IClockFactory {
    clockType = “Larry"
    numcreated = 0
    public instance() : IClock {
        this.numcreated++;
        return new Clocks.Clock1}
    public numCreated() {return this.numcreated}
}

class ClockFactory2 implements IClockFactory {
    clockType = “Curly"
    numcreated = 0
    public instance() : IClock {
        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.

# 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: Have a factory that always returns the same clock

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

export default class SingletonClockFactory  {
    private static isInitialized  : boolean = false

    private static theClock : IClock

    public static instance () : IClock {
        if (!(SingletonClockFactory.isInitialized)) {
            SingletonClockFactory.theClock
                = (new ClockFactory).instance()
            SingletonClockFactory.isInitialized = true
        }
        return SingletonClockFactory.theClock
    }
}
```

# Describing your design using these vocabulary words

When I create an object that needs a clock, I get a copy of the master clock from the clock factory, and then I have the 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

- 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