

CS 4530: Fundamentals of Software Engineering

Lesson 1.3 Object-Oriented Design Principles

Jonathan Bell, Adeel Bhutta, Ferdinand Vesely, Mitch Wand
Khoury College of Computer Sciences

Outline of this lesson

1. Reminder:
 - the purposes of the principles
 - Difficulties the principles should help with
2. Five principles for OO systems

Learning Objectives for this Lesson

- By the end of this lesson you should be able to:
 - Describe the purpose of our design principles
 - List five object-oriented design principles and illustrate their expression in code
 - Identify some violations of the principles and suggest ways to mitigate them

The Challenge: Controlling Complexity

- Software systems must be comprehensible by humans
- Why? Software needs to be maintainable
 - continuously adapted to a changing environment
 - Maintenance takes 50–80% of the cost
- Why? Software needs to be reusable
 - Economics: cheaper to reuse than rewrite!

Five Principles for OO Programming

Five Principles for OO Programming

1. Make Your Interfaces Meaningful
2. Depend only on behaviors, not their implementation
3. Keep Things as Private as You Can
4. Favor Dynamic Dispatch Over Conditionals
5. Favor Interfaces Over Subclassing

Make a sticky note with this list, too.

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
- For our purposes today, we don't mean anything else, like how much memory or time a program uses.

Review: TypeScript interfaces

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

```
class CartesianPoint implements AbsPoint {
  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 AbsPoint {
  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).

We have many classes that implement the same interface

- In a kitchen, for example, we might have

```
class Model101Thermometer implements AbsTemperatureSensor {  
    getTemperature () : Temperature {...}  
    ...  
}
```

```
class AmazonCheapThermometerModel12034 implements AbsTemperatureSensor {  
    getTemperature () : Temperature {...}  
    ...  
}
```

```
class VikingRefrigeratorThermometerModel1178 implements AbsTemperatureSensor {  
    getTemperature () : Temperature {...}  
    ...  
}
```

These all probably
work in very
different ways!

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}  
}
```

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

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

Remember: one interface/one job

- Just like one function/one job...
- If you have a class that needs to advertise two sets of behaviors, you can always have it implement two interfaces.
- The fancy name for this is **interface segregation**.

Principle 2: Depend only on behaviors, not their implementation

```
class TemperatureMonitor {  
    constructor(  
        private sensor: AbsTemperatureSensor,  
        private maxTemp: Temperature,  
        private minTemp: Temperature,  
        private alarm: AbsAlarm,  
    ) { }  
  
    // if the sensor is out of range, sound the alarm  
    public checkSensor(): void {  
        let temp: Temperature = this.sensor.getTemperature()  
        if ((temp < this.minTemp) || (temp > this.maxTemp))  
        { this.alarm.soundAlarm() }  
    }  
}  
  
// sounds an alarm  
interface AbsAlarm { soundAlarm(): void }
```

Principle 2: Depend only on behaviors, not their implementation

```
class TemperatureMonitor {
    constructor(
        private sensor: AbsTemperatureSensor,
        private maxTemp: Temperature,
        private minTemp: Temperature,
        private alarm: AbsAlarm,
    ) { }

    // if the sensor is out of range, sound the alarm
    public checkSensor(): void {
        let temp: Temperature = this.sensor.getTemperature()
        if ((temp < this.minTemp) || (temp > this.maxTemp))
        { this.alarm.soundAlarm() }
    }
}

// sounds an alarm
interface AbsAlarm { soundAlarm(): void }
```

Your new Vocabulary Word:

Dependency Inversion



```
class TemperatureMonitor {
    constructor(
        private sensor: AbsTemperatureSensor,
        private maxTemp: Temperature,
        private minTemp: Temperature,
        private alarm: AbsAlarm,
    ) { }

    // if the sensor is out of range, sound the alarm
    public checkSensor(): void {
        let temp: Temperature = this.sensor.getTemperature()
        if ((temp < this.minTemp) || (temp > this.maxTemp))
        { this.alarm.soundAlarm() }
    }
}

// sounds an alarm
interface AbsAlarm { soundAlarm(): void }
```

Another vocabulary word: *Composition*



```
class TemperatureMonitor {
    constructor(
        private sensor: AbsTemperatureSensor,
        private maxTemp: Temperature,
        private minTemp: Temperature,
        private alarm: AbsAlarm,
    ) { }

    // if the sensor is out of range, sound the alarm
    public checkSensor(): void {
        let temp: Temperature = this.sensor.getTemperature()
        if ((temp < this.minTemp) || (temp > this.maxTemp))
        { this.alarm.soundAlarm() }
    }
}

// sounds an alarm
interface AbsAlarm { soundAlarm(): void }
```

Yet another vocabulary word: *Delegation*



```
class TemperatureMonitor {
    constructor(
        private sensor: AbsTemperatureSensor,
        private maxTemp: Temperature,
        private minTemp: Temperature,
        private alarm: AbsAlarm,
    ) { }

    // if the sensor is out of range, sound the alarm
    public checkSensor(): void {
        let temp: Temperature = this.sensor.getTemperature()
        if ((temp < this.minTemp) || (temp > this.maxTemp))
        { this.alarm.soundAlarm() }
    }
}

// sounds an alarm
interface AbsAlarm { soundAlarm(): void }
```


Principle 3: Keep Things as Private as You Can

- In general, you don't know who is using your code
- You don't want people messing with your data.
 - You might have some invariants that your code depends on, and somebody else might come in and break them.
- You don't want people depending on the details of your code.
 - If you change your details, you might break somebody else's code, which would be BAD.

Example (1)

```
// getCounter ()    always returns an even number
// bumpCounter (n) increases the value of the counter
interface Interface1 {
    getCounter () : number
    bumpCounter (n:number) : void
}

class Class1 implements Interface1 {
    private counter = 0
    // INVARIANT: counter is even
    public getCounter() { return this.counter }
    public bumpCounter (n: number): void {
        // the interface didn't say anything about what do with n.
        this.counter = this.counter + 2
    }
}
```

Example (2)

```
class Class2 implements Interface1 {  
    public counter = 0  
    // INVARIANT: counter is even  
    public getCounter() { return this.counter }  
    public bumpCounter (n: number): void {  
        // the interface didn't say anything about what do with n.  
        this.counter = this.counter + 2  
    }  
}
```

```
let o = new Class2();  
o.counter++;  
console.log(o.getCounter()) // prints 1
```

Example (3)

```
class Class2 implements Interface1 {  
    public c = 0  
    // INVARIANT: counter is even  
    public getCounter() { return this.c }  
    public bumpCounter (n: number): void {  
        // the interface didn't say anything  
        // about what do with n.  
        this.c = this.c + 2  
    }  
}
```

```
let o = new Class2();  
o.counter++; // compiler error!  
console.log(o.getCounter()) // prints 1
```

Principle 4: Favor Dynamic Dispatch Over Conditionals

- We'd like to arrange things so that you can extend your system by adding code, rather than changing it.
- We already saw this in the TemperatureSensor example.
- Let's look at another example.

A Tiny Shape-Manipulation System

- We want to represent two kinds of shapes: squares and circles
- All we have to do is compute the area of a shape.

Naïve representation

```
type Shape = Square | Circle

export class Square {
  constructor (public side: number) {}
}

export class Circle {
  constructor (public radius: number) {}
}

export function area (s:Shape) : number {
  if (s instanceof Square) {
    return s.side * s.side
  } else if (s instanceof Circle) {
    return Math.PI * s.radius * s.radius
  }
}
```

Let's add a new kind of shape to the system

```
// represents ncopies of base shape, arranged in a row without overlaps
export class ShapeArray {
  constructor (public base: Shape, public ncopies: number) {}
}
```


We need to modify our existing code to incorporate this

```
type Shape = Square | Circle | ShapeArray

export class Square { ... }
export class Circle { ... }

// represents ncopies of base shape, arranged in a row
export class ShapeArray {
  constructor (public base: Shape, public ncopies: number) {}
}

export function area (s:Shape) : number {
  if (s instanceof Square) {
    return s.side * s.side
  } else if (s instanceof Circle) {
    return Math.PI * s.radius * s.radius
  } else if (s instanceof ShapeArray) {
    return s.ncopies * area(s.base)
  }
}
```

A better idea: use an interface!

```
export interface Shape {  
    area() : number  
}
```

```
export class Square implements Shape {  
    constructor(private side: number) { }  
    area() : number { return this.side * this.side }  
}
```

```
export class Circle implements Shape {  
    constructor (private radius: number) {}  
    area() : number { return Math.PI * this.radius * this.radius}  
}
```

This is "classic" object-oriented design

- Classic OO says: package the operations with the class.

To add a new shape, you just add code

```
// represents ncopies of base shape, arranged in a row
export class ShapeArray implements Shape {
  constructor (public base: Shape, public ncopies: number) {}
  area() : number { return this.ncopies * this.base.area() }
}
```

- No need to modify existing code!

Now `s.area()` works on any shape

- The old code exported **area** as a function, so if we wanted, we could say

```
export function area (s:Shape) : number {  
    return s.area()  
}
```

The new version works exactly like the old version

```
// import {Square, Circle, ShapeArray, area} from './area1'
import {Square, Circle, ShapeArray, area} from './area2'

describe( "tests of area", () => {
  test("test of square", () => {
    expect(area(new Square(2))).toBe(4)
  })

  test("test of circle", () => {
    expect(area(new Circle(2))).toEqual(Math.PI*4)
  })

  test("test of ShapeArray", () => {
    expect(area(new ShapeArray(new Square(2), 3))).toEqual(12)
  })

  // etc
}
```

- We can use the same tests for either one.

Adding new shapes is easy. What about adding new operations?

- Here we knew the operation(s) in advance
- What if we wanted to add new operations to an existing code base
- Need to add a new operation to the interface (easy– all in one place)
- Need to implement the new operation in each class that implements the interface (might be harder– might be scattered across code base.)
- There's a solution to this, called the Visitor Pattern
 - But that's beyond the scope of this lesson.

Another vocabulary word...

- The idea that you can extend your system by adding code, rather than changing it, is called **the open-closed principle**.
- The system is "open" for extension but "closed" for modification.
- This is another vocabulary word for your coop interview.



Principle 5: Favor Interfaces Over Subclassing

- What happened to inheritance (subclassing) in this story?
- An interface specifies some of the **behavior** of the classes that implement it.
- A superclass specifies some of the **algorithms** of the classes that inherit from it.
 - It means that the subclasses (even those that will be added in the future) can see some of the details of your algorithm
 - Exactly what details depend on the programming language; let's see what happens in Typescript

Example: Clocks

```
export default interface AbsClock {  
  
    // sets the time to 0  
    reset():void  
  
    // increments the time  
    tick():void  
  
    // returns the current time  
    getTime():number  
}
```

Some implementations of AbsClock

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

```
export class Clock1 implements AbsClock {  
  private time = 0  
  public reset () {this.time = 0}  
  public tick () { this.time++ }  
  public getTime(): number {  
    return this.time  
  }  
}
```

// counts down from 0

```
export class Clock2 implements AbsClock {  
  private time = 0  
  public reset () {this.time = 0}  
  public tick () { this.time-- }  
  public getTime () {return (0 - this.time)}  
}
```

// counts up from 42

```
export class Clock3 implements AbsClock {  
  private time = 42  
  public reset () {this.time = 42}  
  public tick () { this.time++ }  
  public getTime () {return this.time - 42}  
}
```

Implementations all
different!

Use inheritance only when there is shared implementation. Example:

Three Implementations of AbsClockFactory

```
interface AbsClockFactory {
    instance() : AbsClock
    clockType : string
    numCreated() : number
}

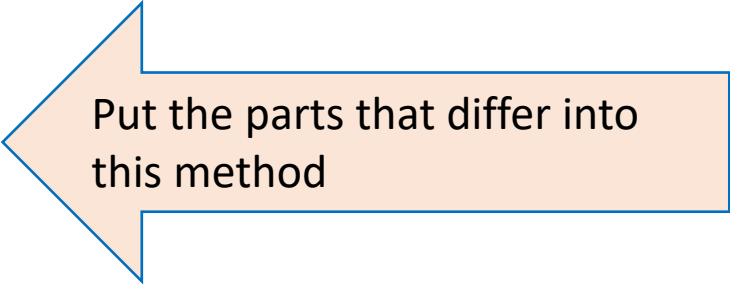
class ClockFactory1
implements AbsClockFactory {
    clockType = "Clock1"
    numCreated = 0
    public instance() : AbsClock {
        this.numCreated++;
        return new Clocks.Clock1}
    public numCreated() {
        return this.numCreated
    }
}
```

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

class ClockFactory3 implements AbsClockFactory {
    clockType = "Clock3"
    numCreated = 0
    public instance() : AbsClock {
        this.numCreated++;
        return new Clocks.Clock3}
    public numCreated() {
        return this.numCreated
    }
}
```

Factor Out Common Portions of Implementation Into a Superclass

```
abstract class ClockFactorySuperClass implements AbsClockFactory
{
    abstract clockType: string
    protected abstract buildClock() : AbsClock
    protected numCreated = 0
    public instance() : AbsClock {
        this.numCreated++;
        return this.buildClock()
    }
    public numCreated() {return this.numCreated}
}
```



Put the parts that differ into
this method

Subclasses implement only the parts that vary

```
class ClockFactory1AsSubclass extends ClockFactorySuperClass
  implements AbsClockFactory {
    clockType = "Clock1"
    protected buildClock() : AbsClock {
      return new Clocks.Clock1}
  }
class ClockFactory2AsSubclass extends ClockFactorySuperClass
  implements AbsClockFactory {
    clockType = "Clock2"
    protected buildClock() : AbsClock {
      return new Clocks.Clock2}
  }
```

That completes
our five
principles



Whose principles are these?

- There are lots of lists of principles out there.
- These are ours.
- One list you should know is **SOLID**. This is an acronym for:
 - S: Single Responsibility
 - O: Open/Closed Principle
 - L: Liskov substitution principle (this has to do with inheritance, so it's not so important for us right now.)
 - I: Interface Segregation
 - D: Dependency Inversion
- So we've covered 4 out of 5 of these.

Another set of principles

- Abstraction
 - Encapsulation
 - Modularity
 - Hierarchy
-
- These are properties that good code should have; we're more interested in what you need to do in order to write good code in the first place.

Review: Learning Objectives for this Lesson

- You should now be able to:
 - Describe the purpose of our design principles
 - List five object-oriented design principles and illustrate their expression in code
 - Identify some violations of the principles and suggest ways to mitigate them