CS 4350: Fundamentals of Software Engineering
CS 5500: Foundations of Software Engineering

Lesson 1.3 Object-Oriented Design Principles

Jon Bell, John Boyland, 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 5 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!

The Challenge: Controlling Complexity

- How? Make programs readable.
- How? Make programs flexible.
- How? Make programs modular.

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 larger, like how much memory or time a program uses.

Interfaces are where we specify behaviors

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

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

interface TemperatureSensor {
    // return 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).

Note that we've specified what these numbers MEAN (see Principle 2 from the last Lesson)

Might we want to put other methods in ITemperatureSensor? Maybe we want it to report its location, too! Why might or might not this be a good idea?

We have many classes that implement the same interface

- In a kitchen, for example, we might have

```
class RefrigeratorThermometer implements TemperatureSensor {
    getTemperature () : Temperature {...}
    ...
}

class OvenThermometer implements TemperatureSensor {
    getTemperature () : Temperature {...}
    ...
}

class CandyThermometer implements TemperatureSensor {
    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 TemperatureSensor. For example:

```
class NotReallyASensor implements TemperatureSensor
{
    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 TemperatureSensor.

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.

Look it up! You should look up each of these vocabulary words on the internet so you will be prepared to define them if your coop interviewer asks you!

Principle 2: Depend only on behaviors, not their implementation

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

    // 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 IAlarm { soundAlarm(): void }
```

Review: TypeScript classes

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

class CartesianPoint implements Point {
    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 Point {
    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...

Principle 2: Depend only on behaviors, not their implementation

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

    // 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 IAlarm { soundAlarm(): void }
```

The monitor doesn't care what kind of TemperatureSensor it's hooked up too. It only cares that it's a correct TemperatureSensor, i.e., that sending it a getTemperature message will return with the temperature at the sensor's location.

Similarly, it doesn't care what kind of alarm it's hooked up to– only that sending the alarm a soundAlarm message will cause an alarm to sound.

Principle 2: Depend only on behaviors, not their implementation

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

    // 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 IAlarm { soundAlarm(): void }
```

This example also illustrates one class/one job. There are three classes here:

1. The sensor senses the temperature
2. The monitor checks to see if the temperature is out of range, and tells the alarm to sound if it is.
3. The alarm actually sounds the alarm.

Your new Vocabulary Word

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

    // 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 IAlarm { soundAlarm(): void }
```

Vocabulary Word: this Principle is called Dependency Inversion. This is a fancy word you can use to impress your coop interviewer.

Another vocabulary word: *Composition*

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

    // 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 IAlarm { soundAlarm(): void }
```

Giving one class a reference to an object of another class (or interface) is sometimes called Composition. That's another vocabulary word you should know for your coop interview.

Delegation is using Composition to avoid hard work

```
interface IWorker {
    // PURPOSE: ....
    doTheHardWork(n:number): void
}

class Class1 {
    constructor(worker: IWorker) { }
    public doTheClass1Task(n:number): void {
        ...
        worker.doTheHardWork(n+39)
        ...
    }
    public anotherAMethod() { }
}
class Class2 {
    constructor(worker: IWorker) { }
    public doTheClass2Task (n:number): void {
        ...
        worker.doTheHardWork(n-5)
        ...
    }
```

Vocabulary Word: Delegation.

Here Class1 and Class2 both delegate their hard work to 'worker'. They don't care how 'worker' is implemented, only that it satisfies the purpose described by Iworker.

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 (BAD!)

Vocabulary word: this idea is called encapsulation.

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
    }
}
```

This is good. Nothing can ever cause getCounter() to return an odd number.

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.bumpCounter();
o.counter++;
console.log(o.getCounter) // prints 3
```

Oh no! We've reached inside Class2 and caused getCounter() to become odd.

Not only that, but now it seems that Class2 is not really an implementation of Interface1 !

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.bumpCounter();
o.counter++;                    // compiler error here
console.log(o.getCounter)
```

when we wrote 'public counter' we announced the name 'counter' for the world to use, just like the names 'getCounter' and 'bumpCounter'. So if we change that name, we'll break all the code that uses it.

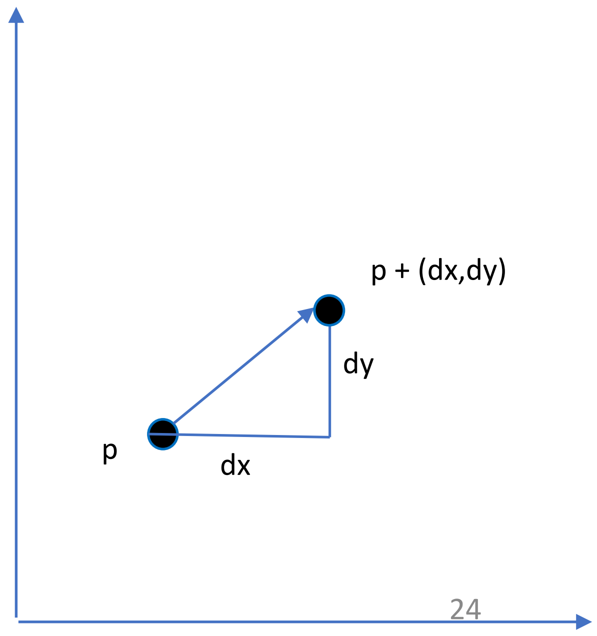
For example, this code depends on the name 'counter' (for better or worse!). Whatever it used to do, it's now entirely broken and needs to be rewritten.

Principle 4: Favor Dynamic Dispatch Over Conditionals

- We already saw a flavor of this in the income-tax example.
- Let's look at another example.

A Tiny Shape-Manipulation System

- Represent three kinds of shapes:
  - circle,
  - square
  - compound of two shapes
- Each shape exists at a particular position on the screen
- The system must support 2 operations on shapes
  - weight : Shape -> number
    - RETURNS: the weight of the given shape, assuming that each shape weighs 1 gram per pixel of area
  - translate : Shape, dx, dy -> Shape
    - Returns a shape like the original, but translated by (dx, dy)



Solution with conditionals (1)

```
type Shape = Circle | Square | Compound
// radius and side in pixels, must be >= 0
type Circle =   { type: "Circle", pos: ScreenPosition, radius: number }
type Square =   { type: "Square", pos: ScreenPosition, side: number }
type Compound = { type: "Compound", front: Shape, back: Shape }

// return weight of the shape, assuming each shape weighs
// 1 gram per pixel of area.
function weightOfShape(s: Shape): number {
    switch (s.type) {
        case "Circle":
            { return (Math.PI * s.radius * s.radius); }
        case "Square":
            { return s.side * s.side }
        case "Compound":
            { return weightOfShape(s.front) + weightOfShape(s.back) }

    }
}
```

Solution with conditionals (2)

```
// returns a shape like the original, but translated by dx, dy
function translateShape(s:Shape, dx:number, dy:number):Shape {
    switch (s.type) {
        case "Circle":
            { return {type: "Circle", pos: translatePosition(s.pos,dx,dy),
                      radius: s.radius} }
        case "Square":
            { return {type:"Square", pos: translatePosition(s.pos,dx,dy),
                      side: s.side} }
        case "Compound":
            { return {
                type:  "Compound",
                front: translateShape(s.front, dx, dy),
                back:  translateShape(s.back, dx,dy)
        }}

    }
}
```

What's more likely to change?

- There will be more new functions, but the set of shapes will be the same
  - Then this solution is pretty good– you can always add more functions to the system
- The set of shapes is likely to differ a lot, but the set of functions will be pretty much the same
  - Yuck! You'll need to go through and change the code in each of the functions

Interfaces to the rescue!

```
// a Shape is anything that has a weight method and a translate method
// that have the right meaning.
// MEANING OF WEIGHT AND TRANSLATE GOES HERE...
interface Shape {
    weightOfShape () : number,
    translateShape(dx:number, dy:number) : Shape
}
```

Represent each shape as a class implementing the Shape interface

```
// radius in pixels, must be >= 0
class Circle implements Shape {
    constructor (
        private pos: ScreenPosition,
        private radius: number
    ) { }
    public weightOfShape () : number  { return (Math.PI * this.radius * this.radius) }
    public translateShape (dx:number, dy:number) : Circle {
        return new Circle(
            translatePosition(this.pos, dx, dy),
            this.radius
        )
    }
}
```

Represent each Shape as a class (2)

```
// side in pixels, must be >= 0
class Square implements Shape {
    constructor (private pos:ScreenPosition, private side:number) {}
    public weightOfShape () : number {return this.side * this.side}
    public translateShape (dx:number, dy:number) : Square {
        return new Square(
            translatePosition(this.pos, dx, dy),
            this.side
        )
    }
}
```

Represent each Shape as a class (3)

```
class Compound implements Shape {
    constructor(private front:Shape, private back:Shape){}
    public weightOfShape (): number {
        return this.front.weightOfShape() + this.back.weightOfShape()
    }
    public translateShape (dx: number, dy: number) {
        return new Compound (
            this.front.translateShape(dx, dy),
            this.back.translateShape(dx, dy)
        )
    }
}
```

This is "classic" object-oriented design

- Let's look at this graphically...

Original vs. OO organization

Here's another way of visualizing the same thing. Here we have six small rectangles corresponding to our six pieces of functionality.

Original:	Square	Circle	Compound
weight			
translate			

OO:	Square	Circle	Compound
weight			
translate			

In the original organization, all the pieces corresponding to **weight** are written together (symbolized here by outlining them in red), and all the pieces corresponding to **translate** are written together (outlined in green).

In the object-oriented organization, all the pieces for **square** are written together (the orange outline in the lower table), all the pieces for **circle** are written together (the green outline), and all the pieces for compound are written together (the brown outline).

Adding a New Data Variant

If we add a new kind of data, such as a triangle, what will we need to change?

We will need 2 pieces of code: one to compute the weight of a triangle and one to translate it

Original:	Square	Circle	Compound
weight			
translate			

OO:	Square	Circle	Compound
weight			
translate			

In the original organization, the two cells correspond to different portions of our file, so we will need to edit two pieces of our file: the **weight** function and the **translate** function.

In the object-oriented organization, we will add the two pieces in a single place in our file: the new **triangle** class.

Adding a New Operation

Original:	Square	Circle	Compound
weight			
translate			
rotate	new code 1	new code 2	new code 3

OO:	Square	Circle	Compound
weight			
translate			
rotate	new code 1	new code 2	new code 3

If we add a new operation such as **move**, what needs to change?

In the original organization, we add the new code in a single function definition, the function **rotate**, symbolized by the blue outline above.

In the object-oriented organization, we must add a **rotate** method in each of our classes.

Extensibility

	Original Org.	O-O Org.
New Data Variant	requires editing in many places	all edits in one place
New Operation	all edits in one place	requires editing in many places

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.

What's the tradeoff?

- Object-oriented organization is better when new data variants are more likely than new operations.
- The original organization is better when new operations are more likely than new data variants.
- In the real world, you may not have a choice:
  - this decision is up to the system architects
  - or may need compatibility with an existing system
- There are ways to get the best of both worlds
  - but these are beyond the scope of this course

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:

```
// 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 {
    protected counter = 0
    // INVARIANT: counter is even
    public getCounter() { return this.counter }
    public bumpCounter (n: number): void { this.counter = this.counter + 2 }
}
```

Here's our old friend Class1. This time we've made 'counter' protected, meaning that it's only visible to the subclasses.

But a subclass can do as much damage as anyone else. Here Class2 can violate the invariant.

tl;dr: subclassing weakens encapsulation!

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.

Review: Learning Objectives for this Lesson

- You should now be able to:
  - Describe the purpose of our design principles
  - List 5 object-oriented design principles and illustrate their expression in code
  - Identify some violations of the principles and suggest ways to mitigate them

Whew! That was a big chunk of stuff. Sorry about that, but we want to get you started on the right foot. You can find lots of more information in the recommended textbooks and on the internet.

Next steps…

- Formulate some questions and come to the class meeting!
- Next week, we'll learn about how to organize and document your code when you have more classes than our examples so far.