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

Lesson 1.2 General Program Design Principles

Jon Bell, John Boyland, Mitch Wand

Khoury College of Computer Sciences

Outline of this lesson

1. The purposes of the principles
2. Difficulties the principles should help with
3. Five general-purpose principles
   - usable for all programming, not just object-oriented

In the next lesson, we'll present five more principles that are specific to object-oriented programming

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 general 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

- Software systems must be comprehensible by humans
  - How? Make programs readable.
  - How? Make programs flexible.
  - How? Make programs modular.

The biggest obstacle: coupling

- Two pieces of code are *coupled* if a change in one demands a change in the other.
- A coupling represents an agreement between the two pieces of code.
  - They may agree on:
    - names
    - order (e.g. of arguments)
    - meaning (e.g. meaning of data)
    - algorithms
- The more two pieces of code are coupled, the harder they are to understand and modify: you have to understand both to understand either of them.

There's a fancy word for this: *connascence* (meaning "born together")

More coupling means less readability, less modifiability

Five general-purpose principles

Five General Principles
1. Use Good Names
2. Design Your Data
3. One method/one job
4. Don't Repeat Yourself
5. Don't Hardcode Things That Are Likely To Change

Good idea: make a sticky note with this list and keep it on your laptop screen.

Principle 1. Use Good Names

- The name of a thing is a first clue to the reader about what the thing means.
  - often, it's the only clue ☹
- Use good names for
  - constants
  - variables
  - functions/methods
  - data types

Good Names for Constants

- Replace magic numbers with good names

```
let salesprice = netPrice * 1.06
```

Where did that 1.06 come from?

```
const salesTaxRate = 1.06
let salesPrice = netPrice * salesTaxRate
```

Oh, it's the sales tax? Are there many occurrences of that 1.06 in your code? (Probably!) Will the sales tax rate ever change? (Probably!)

Let's fix it!

But use good names!

```
int a[100]; for (int i = 0; i <= 99; i++) a[i] = 0;
```

Even if you search for 100, you'll miss the 99!

```
int SIZE = 100;
int a[SIZE]; for (int i = 0; i <= SIZE-1; i++) a[i] = 0;
```

```
int ONE_HUNDRED = 100;
int a[ONE_HUNDRED]; ...
```

No.....!

But use GOOD names!

Good Names for Variables and Types

```
var t : number
var l : number
```

What do these variables mean?

Better names would give the reader a clue.

```
var temp : number
var loc  : number
```

Does 'temp' mean 'temporary', or 'temperature', or something else?

```
var temp : Temperature
var loc  : SensorLocation
```

Good names for the data types solves the problem.

Good Names for Functions and Methods

```
function checkLine () : boolean
```

What are you checking it for? Length? Illegal Syntax? or what?

```
function isLineTooLong () : boolean
```

Ahh, now we know!

Good Names for Functions and Methods

- Use noun-like names for functions or methods that return values, e.g.

```
let c = new Circle(initRadius)
let a = c.diameter()
```

- not:

```
let a = c.calculateDiameter()
```

- Reserve verb-like names for functions or methods that perform actions, like

```
table1.addItem(student1,grade1)
```

Your workplace should have coding standards for things like this. This particular item is part of Prof. Wand's personal coding practice

Principle 2. Design Your Data

- You need to do three things:
  1. Decide what part of the information in the "real world" needs to be represented as data
  2. Decide how that information needs to be represented as data
  3. Document how to interpret the data in your computer as information about the real world

Example:

- Right now I am wearing a red shirt, and I've decided I need to represent that fact in my program.
- How should I represent that in my program?
- I need to represent the color red. Possibilities:
  - "red" (English text)
  - "RED" (English text)
  - "Lāla" (Hindi, according to Google)
  - #ff0000

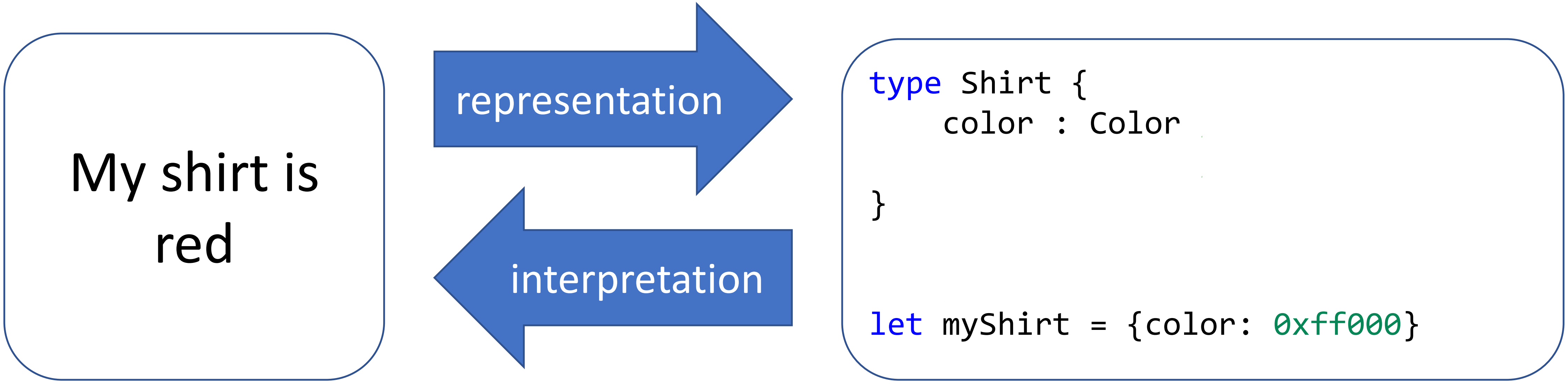
Example (2)

- And of course we also need to represent my shirt.
- In that representation, we have to represent its color.
- Here's one of many possibilities:

```
type Shirt {
    color : Color   // the color of the shirt
}

let myShirt = {color: 0xff000} // my shirt
```

The Big Picture



- How do we know that these are connected?
- Answer: we have to write down the interpretation
- In our Typescript infrastructure, we do that with the comments.

Principle 3: One Method/One Job

- Each class, and each method of that class, should have one job, and only one job
- If your method has more than one job, split it into 2 methods. Why?
  - You might want one part but not the other
  - It's easier to test a method that has only one job
- You call both of them if you need to.
  - or write a single method that calls them both
- Same thing for classes.

The fancy name for this is "The Single Responsibility Principle". You can use this if you want to impress your coop interviewer.

Principle 4: Don't Repeat Yourself

- If you have some quantity that you use more than once, give it a name and use the name.
- That way you only need to change it in one place!
- And of course you should use a good name
- If you have some task that you do in many places, make it into a procedure.
- If the tasks are slightly different, turn the differences into parameters.

We saw this before with the sales tax and array bound examples.

This is called "Single Point of Control"

A real example

```
function testequal <T> (testname: string, actual: T, correct: T) {
    it(testname,
        function () { assert.deepEqual(actual, correct) })
}

describe('tests for count_local_morks', function () {
    testequal('empty crew',count_local_morks(ship1),0)
    testequal('just Mork',count_local_morks(ship2),1)
    testequal('just Mindy',count_local_morks(ship3),0)
    testequal('two Morks',count_local_morks(ship4),2)
    testequal('drone has no Morks',count_local_morks(drone1),0
})
```

Think of how much typing this saves!

Plus, if I ever need to change what testequal does, I can do it all in one place.

Principle 5: Don't Hardcode Things That Are Likely To Change

- "No magic numbers" and "Don't Repeat Yourself" are already examples of this.
- General strategy: If there something that might change, give it a name
  - if it's not already a "thing", refactor to make it a "thing"
  - many strategies for this; let's look at one of them

Example

- Imagine we are computing income tax in a state where there are four rates:
  - One on incomes less than $10,000
  - One on incomes between $10,000 and $20,000
  - One on incomes between $20,000 and $50,000
  - One on incomes greater than $50,000
- You might write something like

You might write something like

```
function grossTax(income: number): number {
    if ((0 <= income) && (income <= 10000)) { return 0 }
    else if ((10000 < income) && (income <= 20000))
    { return 0.10 * (income - 10000) }
    else if ((20000 < income) && (income <= 50000))
    { return 1000 + 0.20 * (income - 20000) }
    else { return 7000 + 0.25 * (income - 50000) }
}
```

This also violates one function/one job: it finds the right bracket AND calculates the appropriate tax

- What might change?
  - The boundaries of the tax brackets might change
  - The number of brackets might change

Not so terrible..

Ouch! Do you really want to dive into that conditional?

So let's represent our data differently

```
// defines the tax bracket for income lower < income <= upper.
// if upper is null, then lower < income  (no upper bound)
type TaxBracket = {
    lower: number,
    upper: number | null,
    base : number
    rate : number
}

let brackets : TaxBracket[] = [
    {lower:0,     upper:10000, base:0, rate:0},
    {lower:10000, upper:20000, base:0, rate:0.10},
    {lower:20000, upper:50000, base:1000, rate:0.20},
    {lower:50000, upper: null, base:7000, rate:0.25}
]
```

The brackets are now a "thing". All the data is in one place, so we have a Single Point of Control

And now it's easy to rewrite our function

```
// defines the incomes covered by a bracket
function isInBracket(income:number, bracket:TaxBracket) : boolean {
    if (bracket.upper == null)
    { return (bracket.lower <= income) }
    else
    { return ((bracket.lower <= income) && (income < bracket.upper))}
}

function taxByBracket(income:number,bracket:TaxBracket) : number {
    return bracket.base + bracket.rate * (income - bracket.lower)
}

function grossTax2 (income:number, brackets: TaxBracket[] ) : number {
    return taxByBracket(income,income2bracket(income,brackets))
}
```

And we are back to one function/one job.

Review: Learning Objectives for this Lesson

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

Next...

- In our next lesson, we'll learn about five more basic principles that are specific to an object-oriented setting.