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

Lesson 2.3 Design Patterns

Jon Bell, John Boyland, Mitch Wand

Khoury College of Computer Sciences

Outline of Lessons 2.3 and 2.4

- Introduction to patterns:
  - what are they?
  - what are they good for?
- Review of patterns you probably know
  - Adapter
  - Composite
  - Iterator
- Some patterns you may or may not know
  - Singleton
  - Observer (sometimes called Listener, or Publish/Subscribe)
  - Visitor

Learning Objectives for this Lesson

- By the end of this lesson you should be able to:
  - Define what a design pattern is and the role it plays in the Software Development process
  - Explain and illustrate the following patterns
    - Adapter
    - Composite
    - Iterator

What is a design pattern?

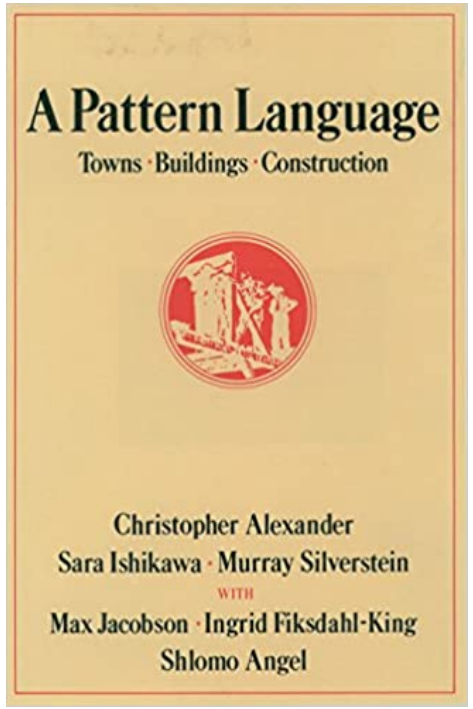
- Think of it as advice from a master to a novice.
  - A master chef may advise a novice on knife technique
  - A golf pro may advise a novice about their swing
  - A piano teacher may advise a student about their posture, or how to interpret a piece
- Often these pieces of advice are stylized and recorded
  - eg "keep your elbow straight" (golf) "use the tip of your knife as a fulcrum" (knife technique)
  - Maybe in a book of "technique"
  - Maybe on YouTube
  - etc.

What is in such a piece of advice?

- A problem to be solved
  - "the golf ball keeps flying off to the side"
  - "it's taking too long to chop the carrots"
- A technique or method for solving the problem
- The technique always needs to be adapted to the problem at hand
  - is the golf ball lying on a slope? what kind of slope?
  - do you have a proper chopping board? what kind of knife are you using?

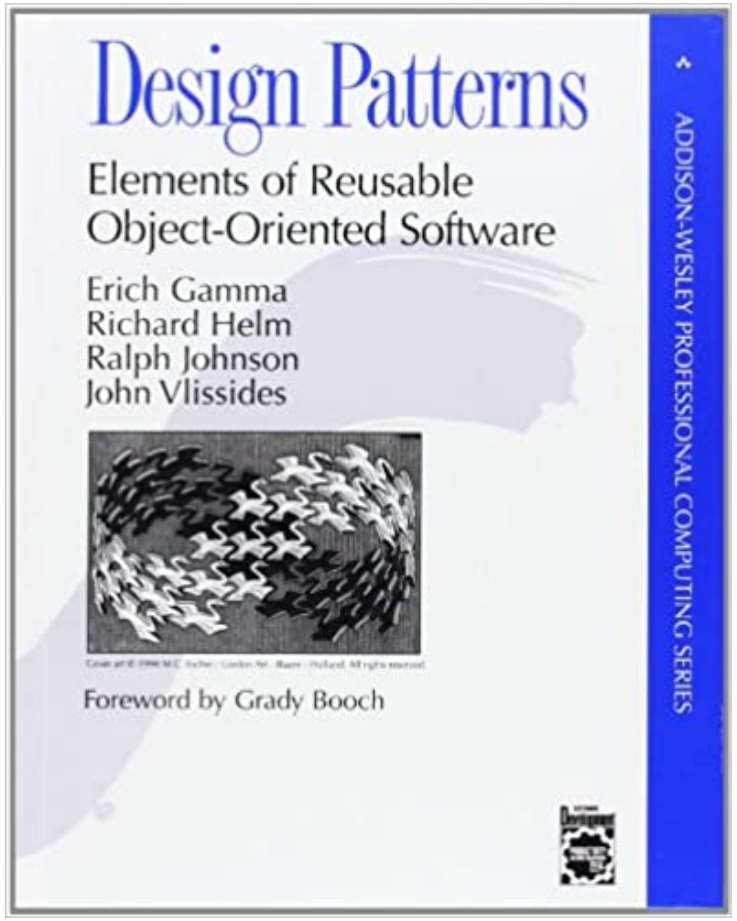
Design Patterns in Architecture

- "A Pattern Language: Towns, Buildings, Contruction" by Christopher Alexander (1977)
- introduced this idea to a wide community beyond architects



Gang of Four Book

- First (and only!) edition 1994
- Introduced this idea to object-oriented design
- Started the "Software Patterns" movement
- Still #1 on Amazon in Object-Oriented Software Design



Definition

- Alexander says:

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

Elements of a Pattern

- the pattern name
- the problem (when to apply the pattern)
- the solution (describes the elements that make up the pattern)
- the consequences (the results and trade-offs of applying the pattern)

This is the official definition, taken from the GoF book.
Alas, when they get around to describing patterns, their descriptions rarely match this outline ☹

Design Patterns are Controversial

- For the last 25 years, software experts have lined up either as pattern fans or pattern skeptics
- Sometimes there are endless debates about whether a given piece of code is or is not an instance of a particular pattern.
- We are just not going to get into that.
  - Take a chill pill!
  - But keep your 5004/5010 notes close at hand.
- These patterns are tools in your toolbox.

Design Patterns are Everywhere

- Everytime you read a blog post or web page with some code illustrations, you are using a design pattern:
  - a piece of code to solve a particular problem
  - and which needs to be adapted to your particular situation.
- But some patterns are classics that have names that you should be familiar with.

Problem #1:

- Suppose we need to implement a stack, which has the following interface:

```
// the usual stack operations
interface IStack<T> {
  push(t: T) : void
  pop() : T
  size() : number
}
```

- But we have a class **List** that implements **IList:**

```
interface IList<T> {
  // add to end of list
  add(t:T): void
  // remove last element of the list
  remove() : T
  // returns the number of elements in the list
  size() : number
}
```

Of course, in Typescript you'd never do this, because in Typescript we almost always use arrays to represent lists.

Solution: an Adapter

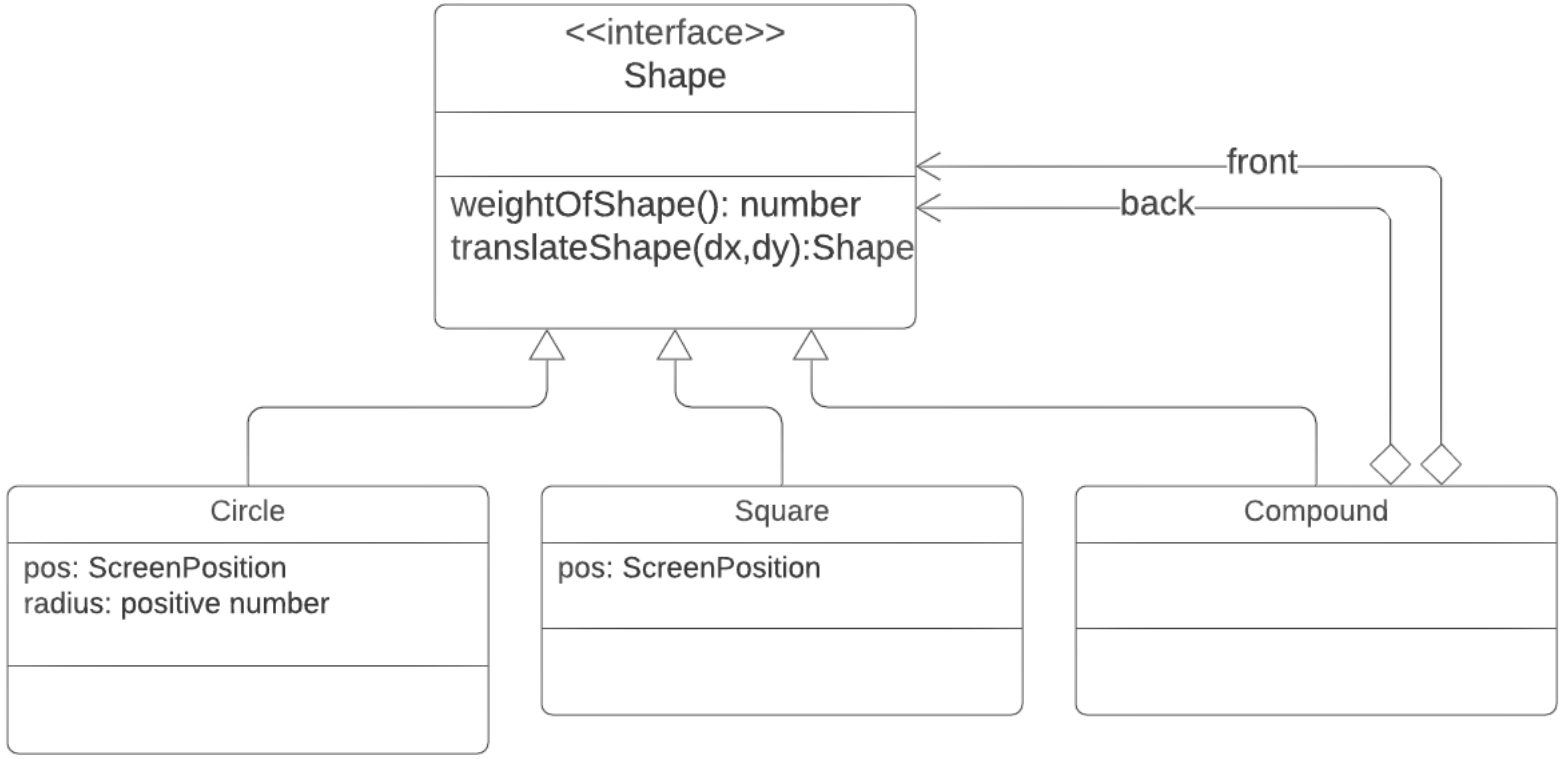
```
class Stack<T> implements IStack<T> {

  // top of stack is at end of list
  constructor (private payload: IList<T>) {}

  public push(t: T): void {
    this.payload.add(t);
  }
  public pop(): T {
    return this.payload.remove();
  }

  public size(): number {
    return this.payload.size()
  }

}
```

Important: if you do something like this, be sure to explain how the list should be interpreted as a stack. (Remember Design Principle 2!)

Problem #2:

- You need to represent data that is tree-like
- Example:
  - shapes, from Week 1.
  - A Shape is either
    - a square
    - a circle
    - or a compound of two shapes: a front shape and a back shape.

Solution: the Composite Pattern



We explained this example in Week 1. Now we've just added a name for what you already know.

Notice the circular dependency between Shape and Compound. That comes along with hierarchical (tree-like) data. There's no avoiding it.

Problem #3:

- You need to systematically go through the elements of some collection.
- Solution 1: Implement your collection using a type that natively supports it.
  - in TypeScript, this typically means an array (a list) or Map
  - These are called *internal iterators*

```
const mylist : Shape[] = ...
mylist.map(shape => ...)
mylist.filter(shape => ...)
mylist.forEach(shape => ...)

for (s in mylist) {...}
```

The function that you apply to each element of the array is called the callback.

Internal iterators like these replace almost all loops in TypeScript. If you are not familiar with them, go look them up.

TypeScript also allows iterators over Maps

```
type StudentTableOut = Map<StudentId,StudentDataOut>

function countAllBins (studentMasterTable: StudentTableOut) {
    let histo = [0,0,0]   // a histogram with 3 bins
    for (let student of studentMasterTable.keys()) {
        let data = studentMasterTable.get(student)
        for (let question of data.keys()) {
            let questionData = data.get(question)
            let bin = questionData.bin
            histo[bin] += ...
        }
    }
    return histo.map(n => n/(histo[0]+histo[1]+histo[2]))
}
```

Solution 2: Define an external iterator

- In TypeScript, there are ways of creating iterators that integrate with things like **for**.
- Alas, these are way too complicated for us to do right now.

The Iterator Pattern

- This is all quite a bit different from what's in the GoF book
- This illustrates how patterns are dependent on the programing language you are working in
- Much of the complexity of the pattern has now been absorbed into the programming language (even in different versions of JavaScript!)

Review: Learning Objectives for this Lesson

- At this point you should be able to:
  - Define what a design pattern is and the role it plays in the Software Development process
  - Explain and illustrate the following patterns
    - Adapter
    - Composite
    - Iterator

Next steps...

- In our next lesson, we will go on to explore three more patterns that you may or may not be familiar with:
  - Singleton
  - Listener
  - Visitor