Principles of Software Construction: Objects, Design, and Concurrency

Refactoring & Anti-patterns

Christian Kästner **Vincent Hellendoorn**

HW3 Feedback

https://rb.gy/xpnh1b

Today

- Midterm debrief
  - Discussing the Decorator Pattern
- Revisiting composition + delegation
- More on inheritance
- Refactoring and Anti-patterns

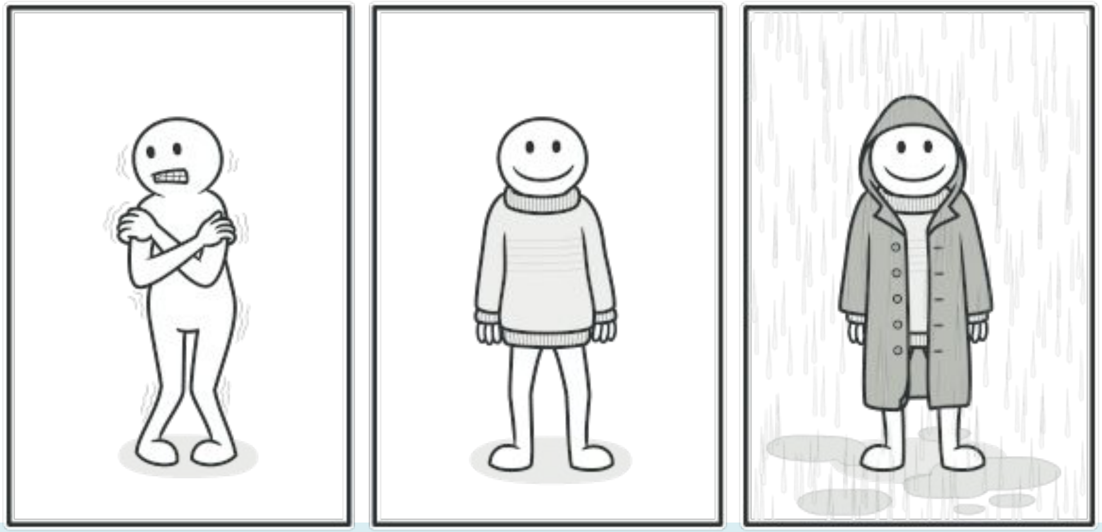
Midterm Debrief

How did it go? Anything deserve discussion?

The Decorator Pattern

You have a complex drawing that consists of many shapes and want to save it. Some logic of the saving functionality is always the same (e.g., going through all shapes, reducing them to drawable lines), but others you want to vary to support saving in different file formats (e.g., as png, as svg, as pdf). You want to support different file formats later.

Why is this not:



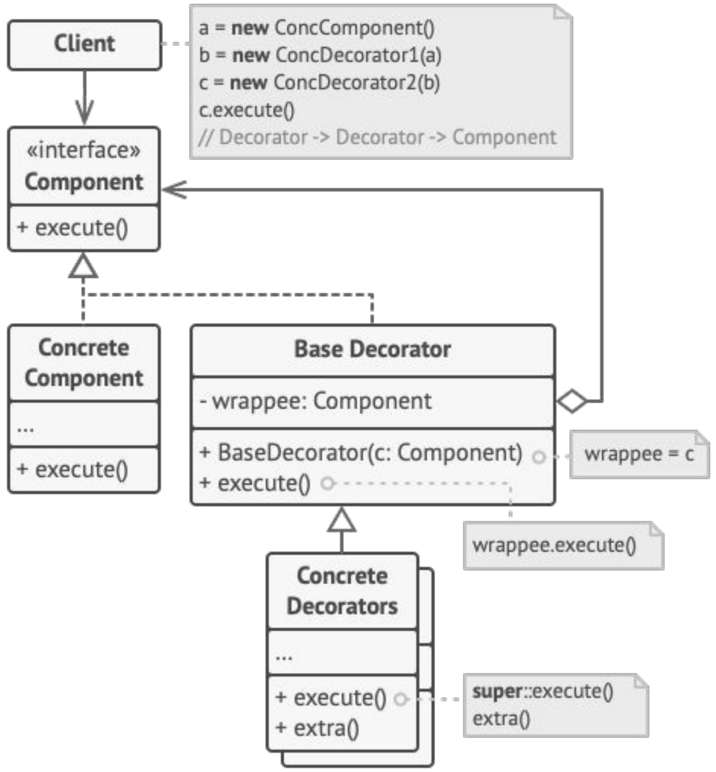
Drawing Example -- Basics

```
class Line {
    // TODO
}

interface Shape {
    toLines(): Line[];
}

class Triangle implements Shape {
    public toLines(): Line[] {
        return ...;
    }
}
```

Drawing Example -- Basics

```
// A drawing consists of many shapes.
class Drawing {
    shapes: Shape[]
    constructor(shapes: Shape[]) {
        this.shapes = shapes;
    }
    public toLines() {
        let lines: Line[] = []
        for (let shape of this.shapes) {
            lines.push(shape.toLines());
        }
        return lines;
    }
}
```

Drawing Example -- Decorator?



```
a = new ConcComponent()
b = new ConcDecorator1(a)
c = new ConcDecorator2(b)
c.execute()
// Decorator -> Decorator -> Component
```

```
super::execute()
extra()
```

Drawing Example -- Decorator?

```
interface DrawingSaver {
    saveDrawing(drawing: Drawing, path: string): void;
}

class BasicSaver implements DrawingSaver {
    public saveDrawing(drawing: Drawing, path: string): void {
        let lines: Line[] = drawing.toLines();
        // Now what?
    }
}
```

Drawing Example -- Decorator?

```
class DrawingSaverDecorator implements DrawingSaver {
    wrappee: DrawingSaver
    constructor(source: DrawingSaver) { this.wrappee = source; }

    public saveDrawing(drawing: Drawing, path: string): void {
        this.wrappee.saveDrawing(drawing, path);
    }
}
class JPEGDecorator extends DrawingSaverDecorator {
    public saveDrawing(drawing: Drawing, path: string): void {
        let lines: Line[] = drawing.toLines();
        // Internally store in JPEG
        super.saveDrawing(drawing, path);
    }
}
```

Drawing Example -- Strategy

```
interface LineFormatter {
    write(lines: Line[], writer: Writer): void;
}

class DrawingSaver {
    public save(drawing: Drawing, formatter: LineFormatter, path: string) {
        let lines: Line[] = drawing.toLines();
        let writer: Writer = new Writer(path);
        formatter.write(lines, writer);
    }
}

class JPEGFormat implements LineFormatter {
    public write(lines: Line[], writer: Writer) { // Store JPEG data. }
}
```

Drawing Example -- Template Method

```
abstract class DrawingSaver {
    public save(drawing: Drawing, path: string) {
        let lines = drawing.toLines();
        let formatted = this.toFormat(lines);
        let writer: Writer = new Writer(path);
        writer.write(formatted);
    }

    abstract toFormat(lines: Line[]): any[];
}

class JPEGSaver extends DrawingSaver {
    public toFormat(lines: Line[]): any[] { // Store JPEG data. }
}
```

Today

- Midterm debrief
  - Discussing the Decorator Pattern
- **Revisiting composition + delegation**
- More on inheritance
- Refactoring and Anti-patterns

Delegation

- *Delegation* is simply when one object relies on another object for some subset of its functionality
  - e.g. here, the `Sorter` is delegating functionality to some `Order`
- Judicious delegation enables code reuse

```
interface Order {
  boolean lessThan(int i, int j);
}
final Order ASCENDING =  (i, j) -> i < j;
final Order DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Order cmp) {
  …
  boolean mustSwap =
    cmp.lessThan(list[i], list[j]);
  …
```

Using delegation to extend functionality

- Consider the `java.util.List` (excerpted):

```
public interface List<E> {
    public boolean add(E e);
    public E       remove(int index);
    public void    clear();
…
}
```

- Suppose we want a list that logs its operations to the console…

Using delegation to extend functionality

The LoggingList *is composed of* a List, and delegates (the non-logging) functionality to that List

```
public class LoggingList<E> implements List<E> {
   private final List<E> list;
   public LoggingList<E>(List<E> list) { this.list = list; }

   public boolean add(E e) {
       System.out.println("Adding " + e);
       return list.add(e);
   }

   public E remove(int index) {
       System.out.println("Removing at " + index);
       return list.remove(index);
   }
}
```

Using inheritance to extend functionality

```
public class LoggingList<E> extends ??? {

   public boolean add(E e) {
       System.out.println("Adding " + e);
       return super.add(e);
   }

   public E remove(int index) {
       System.out.println("Removing at " + index);
       return super.remove(index);
   }
}
```

Using inheritance to extend functionality

```
public class ArrayLoggingList<E> extends ArrayList<E> {

   public boolean add(E e) {
       System.out.println("Adding " + e);
       return super.add(e);
   }

   public E remove(int index) {
       System.out.println("Removing at " + index);
       return super.remove(index);
   }
}
```

The `LoggingList` *is an* `ArrayList`, and relies on it for the (the non-logging) functionality.

Delegation and Design

- Small interfaces with clear contracts
- Classes to encapsulate algorithms, behaviors
    - E.g., the Order

Designing with Inheritance in Mind

- Try to avoid it when composition+delegation is available
  - Delegation reduces coupling
  - Inheritance limits *information hiding*
- Document contracts for inheritance
  - The compiler won't inforce all invariants
- Enforce or prohibit inheritance where possible
  - In Java: `final` & `abstract`

Today

- Midterm debrief
  - Discussing the Decorator Pattern
- Revisiting composition + delegation
- **More on inheritance**
- Refactoring and Anti-patterns

Details: `this`

- Refers to itself, but what is that?
  - In general, behaves as a variable referencing the current object.
  - Knows names of all fields and methods
    - Including private ones, and ones inherited from parents
  - But not generally types, modifiers
    - This is where *reflection* comes in, which is significantly easier in some languages
    - E.g., in Python: `self.__dict__` is all field names on self
    - In JS, `object.__proto__` does something similar
- The definition of ‘this’ can get murky with inheritance
  - Subtle differences between Java and JS/TS, Python

This binding

```
class Parent {
   private int i;
   public Parent() {
       this.i = 5;
   }

   void print() {
       System.out.println(this.i);
   }
}

class Child extends Parent {
   private int i;
   public Child() {
       this.i = 7;
   }
}
```

```
Child m = new Child();

System.out.println(m.i);

m.print();
```

What is ‘this’?

```
class BaseSaver implements DrawingSaver {
   public save(Drawing drawing, Writer writer) {
       Line[] lines = drawing.toLines();
       this.serialize(writer, lines);
   }
   public serialize(writer, lines) { ... }
}

class PNGDrawingSaver implements DrawingSaver
   DrawingSaver delegee;

   public save(Drawing drawing, Writer writer) {
       delegee.save(drawing, writer);
   }
   public serialize(...) { // write PNG }
}
```

Which ‘serialize’ gets called?

```
new PNGDrawingSaver(
  new BaseSaver()).save(...);
```

Details: `super`

- Similar to `this`
- Refers to any (recursive) parent
  - Depending on what is accessed
- In TS, must call super(); before using ‘this’
  - Initializes the class
- In Java, super call needs to be first statement in constructor

Details: `final`

- A final field: prevents reassignment to the field after initialization
- A final method: prevents overriding the method
- A final class: prevents extending the class
  - e.g., `public final class CheckingAccountImpl { …`
- Not present in TypeScript
  - Called “sealed” in some languages

Details: `abstract`

- Method: must be overridden by a non-abstract subclass
- Class: only classes allowed to have abstract members

Details: type-casting

- Sometimes you want a different type than you have
  - e.g., `double pi = 3.14;`
    `int indianaPi = (int) pi;`
- Useful if you know you have a more specific subtype:

```
Account acct = …;
CheckingAccount checkingAcct = (CheckingAccount) acct;
long fee = checkingAcct.getFee();
```

  - Will get a `ClassCastException` if types are incompatible
- Advice: avoid downcasting types
  - Never(?) downcast within superclass to a subclass

In TS:

```
(dog as Animal).identify()
```

Today

- Midterm debrief
    - Discussing the Decorator Pattern
- Revisiting composition + delegation
- More on inheritance
- **Refactoring and Anti-patterns**

Recall: Refactoring and Anti-Patterns

- Often, all the functionality is correct, but the organization is bad
  - High coupling, high redundancy, poor cohesion, god classes, …
- Refactoring is the principal tool to improve structure
  - Automated refactorings even guarantee correctness
    - But you can't always count on those being right
  - A series of refactorings is usually enough to introduce design patterns
- HW4 involves analyzing such a system and making primarily refactoring changes
  - "primarily", because sometimes you do need to alter things slightly.

Anti-patterns

Anti-patterns are *common* forms of bad/no-design

- Can you think of examples?
- Where do they come from?

Anti-patterns

- We have talked a fair bit about bad design heuristics
  - High coupling, low cohesion, law of demeter, …
- You will see a much larger vocabulary of related issues
  - Commonly called code/design “smells”
  - Worth reads:
    - A short overview: https://refactoring.guru/refactoring/smells
    - Wikipedia: https://en.wikipedia.org/wiki/Anti-pattern#Software_engineering
    - Book on the topic (no required reading): Refactoring for Software Design Smells: Managing Technical Debt, Suryanarayana, Samarthyam and Sharma
      - S.O. summary: https://stackoverflow.com/a/27567960

Anti-patterns

- Two ways of looking at this:
  - Design issues that manifest as bad/unmaintainable code
  - Poorly written/evolved code that leads to bad design
  - Next two slides show both

Anti-patterns

- Common system-level anti-patterns
  - Bad encapsulation, violates information hiding
    - public fields should be private; interface leaks implementation details; lack of interface
  - Bad modularization, violates coupling
    - related methods in different places, or vice versa; very large interface; “god” class
  - Bad abstraction, violates cohesion
    - Not exposing relevant functionality; near-identical classes; too many responsibilities
  - Bad inheritance/hierarchy
    - Violating behavioral subtyping; unnecessary inheritance; very large hierarchies (too wide or too deep)

Anti-patterns

- Zooming in: common code smells
  - Not necessarily bad, but worthwhile indicators to check
    - When problematic, often point to design problems
  - Long methods, large classes, and the likes. Suggests bad abstraction
    - Tend to evolve over time; requires restructuring
  - Inheritance despite low coupling (“refused bequest”)
    - Replace with delegation, or rebalance hierarchy
  - ‘instanceof’ (or ‘switch’) instead of polymorphism
  - Overly similar classes, hierarchies
  - Any change requires lots of edits
    - High coupling across classes (“shotgun surgery”), or heavily entangled implementation (intra-class)

Anti-patterns

- Zooming in: common code smells
  - Not necessarily bad, but worthwhile indicators to check
    - When problematic, often point to design problems
  - Excessive, unused hierarchies
  - Operations posing as classes
  - Data classes
    - Tricky: not always bad, but ideally distinguish from regular classes (e.g., ‘record’), and assign responsibilities if any exist (think: FlashCard did equality checking)
  - Heavy usage of one class’ data from another (“feature envy”, “inappropriate intimacy”; poor coupling)
  - Long chains of calls needed to do anything (law of demeter)
  - A class that only delegates work

Anti-patterns

- You can detect them from either side
  - Pick a design principle, look for violations
  - Identify “weird” code and isolate design flaw

Anti-patterns

- You can detect them from either side
  - Pick a design principle, look for violations
  - Identify “weird” code and isolate design flaw
- All fairly easy to spot on their own
  - But in HW4, there are multiple, tangled up
    - We actually provide way more guidance than you’ll get in the wild!
  - How do you approach that?

Refactoring and Anti-patterns

- Identifying multiple design problems
  - Make a list
    - Read the code, record anything that stands out
      - Pay attention to class names and their (apparent) interfaces
      - Make note of repetitive code (esp. across methods)
    - Draw a diagram, using a tool or by hand
      - Spot duplication, (lack of) interfaces, strange inheritance
    - This takes **practice**
  - Don’t solve every problem
    - Many issues are orthogonal
      - Or, at least, you can improve things somewhat
    - When issues intersect, prioritize fixing interfaces

Refactoring

- So where is “refactoring” in all this?
  - It’s what comes next.
  - Most design issues can be resolved with functionality-preserving transformation(s)
    - Too many parameters? Merge relevant ones into object, and/or replace with method calls.
    - Two near-identical classes? Merge their signatures using renamings, parameterization, then delete one or extract super-class
  -

Summary

- Practice applying design patterns, recognizing anti-patterns
  - Create scenarios and try to write code
  - Find examples in public projects
  - We’ll do a case-study on Thursday
- Use this time to gain experience
  - Read lots of code, think about alternatives, like in HW4
  - Learn a vocabulary of anti-patterns (even if imperfect)