Feedback from IEF
Things you like

- Many:
  - In-class discussion
  - Programming assignments
  - Code reviews

- A few
  - The book is useful
  - Use of I-clickers
  - Random tangents
  - Assignments build on each other sometimes
Things you don’t like

■ Many
  ▪ Pop quizzes
  ▪ Book is long winded
  ▪ Subjectivity of some I-clicker questions

■ A few
  ▪ Discussion getting derailed
  ▪ Not each assignment independent
  ▪ Length/difficulty of assignments
  ▪ Strict grading of code reviews
Things to change

- Provide solutions to assignments
- Better feedback on grades
- More discussion of tools
Design Patterns
Design Pattern

“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.” -- Christopher Alexander

Each pattern has 4 essential elements:

- A name
- The problem it solves
- The solution
- The consequences
Let’s start with some “Micro-Patterns” (1)

- **Name:** Most-wanted holder *(or most wanted index)*
- **Problem:** Want to find the “most wanted” element of a collection.
- **Solution:** Initialize most-wanted holder to first element. Compare every other element to value in most-wanted holder, replace if the new value is better.

```java
Thing mostWanted = things[0];
for (int i = 1; i < things.length; i++) {
    if (thing[i].isBetterThan(mostWanted)) {
        mostWanted = thing[i];
    }
}
```

mostWanted = i

Let’s start with some “Micro-Patterns” (2)

- **Name:** One-way flag
- **Problem:** Want to know if a property is true/false for every element of a collection.
- **Solution:** Initialize a boolean to one value. Traverse the whole collection, setting the boolean to the other value if an element violates the property.

```java
boolean allValid = true;
for (Thing thing : things) {
    if (!thing.isValid()) {
        allValid = false; // you could break
    }
}
```
Let’s start with some “Micro-Patterns” (3)

- **Name:** Follower
- **Problem:** Want to compare adjacent elements of collection.
- **Solution:** As you iterate through a collection, set the value of the follower variable to the current element as the last step.

```java
boolean collectionInOrder = true;
Thing follower = null;
for (Thing thing : things) {
    if (follower != null &&
        !thing.isBiggerThan(follower)) {
        collectionInOrder = false;
    }
    follower = thing;
}
```
“Design Patterns” focus on object-level

- Relate to relationships between classes & objects
  - IsA (inheritance) and HasA (containment) relationships

- Many of these seem obvious (in hind sight)
  - The power is giving these names, codifying a best practice solution, and understanding their strengths/limitations.
Problem: Social media updates

- You have your InstaTwitInYouFaceTrest app open and a friend makes a post / updates their status. How do you get the info before the next time you (manually) refresh your app?
The Observer Pattern (a.k.a. Publish/Subscribe)

- **Problem:** Keep a group of objects “in sync” in the presence of asynchronous updates, while minimizing the amount of coupling.

- **Intent:** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

- **Use the Observer pattern when:**
  - When changes to one object requires changes to other and you don’t know which and/or how many.
  - When an object should be able to notify other objects without making assumptions about who these other objects are (i.e., you don’t want these objects tightly coupled).
Class/Object Notation

- Class definitions

Abstract in italics

Methods have parentheses
Variables do not

Types are optional; included when useful
Class/Object Notation (cont.)

- Class relationships

**Diagram: Class/Object Notation**

- Diamond = Has A collection of
- Solid dot = multiple
- Triangle = Inheritance (Is A)
- Dashed line = creates
- Solid line = Has A (containment)
Class/Object Notation (cont.)

- Object instances

Objects have rounded corners
Class/Object Notation (cont.)

- **Interaction Diagram**

  - Solid vertical line = existed before/after interaction
  - Dashed vertical line = didn’t exist
  - Dashed horizontal line = creation

  - Solid horizontal line = invocation
  - Box = period active during interaction
  - Time passing

---

**Figure B.2:** Object diagram

**Figure B.3:** Interaction diagram

Time flowing from top to bottom in interaction diagrams.
Observer Pattern

A) Classes
B) Objects

[Diagram showing the Observer Pattern with classes and methods]
Observer Pattern

A) HasA (containment)
B) IsA (inheritance)
Observer Pattern

- **Solution:**
  - Observers can “attach” to a Subject.
  - When Subject is updated, it calls Update() on all Observers.
  - Observers can query Subject for updated state.
Observer Pattern Interaction Example

- aConcreteObserver modifies a ConcreteSubject
Problem: Navigation systems

- What if we wanted to build a navigation system, but we wanted to support a choice of path finding algorithms (A*, Dijkstra’s algorithm, ...).
Strategy

**Intent:** define a family of algorithms, encapsulate each one, and make them interchangable. Strategy lets the algorithm vary independently from clients that use it.

**Use the strategy pattern when:**

- Many related classes differ only in their behavior.
- You need different variants of an algorithm (e.g., trade-offs)
- An algorithm uses data that clients shouldn’t know about
  - E.g., encapsulate the algorithm data from client
- A class defines multiple behaviors and these are implemented using conditionals.
Strategy Pattern

Solution

- Strategy abstract base class exposes algorithm interface.
- Context object HasA Concrete Strategy object.
- Context object invokes algorithm interface from strategy.
Problem: A*-ing a someone else’s graph

- A friend of yours has a class implementing a different kind of graph and would like to be able to use your A* algorithm on it. Does he have to re-write his code to implement your Astarable interface?
Adapter Pattern

- **Intent:** Convert the interface of a class into another interface that a client expects. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

- **Use the Adapter pattern when:**
  - You want to use an existing class and interface doesn’t match the one that you need
  - You want to create a reusable class that cooperates with unrelated and unforeseen classes (non-compatible interfaces)
  - You need to use several existing subclasses, but it’s impractical to adapt their interface by subclassing every one.
Adapter Pattern

Solution:
- Adapter class IsA derived class of Target type
- Adapter class HasA Adaptee class
- Adapter class delegates requests to Adaptee class