CSC301

Observers, Adapters & Test Doubles
Observer/Observable Pattern

- Common design pattern

- Appeared in the Gang of Four book
  - By Gamma, Helm, Johnson & Vlissides
  - Essentially, a collection of common problems and “recipes” for generic solutions.
Terminology

• A few names:
  ○ Observer-Observable
  ○ Listener
  ○ Publish-Subscribe

• The concept is the same:
  ○ Whenever something happens to object A, object B gets notified and takes an action.
  ○ The two objects care about interfaces (e.g. observer and observable), not concrete implementations.
Design Patterns, GoF
Page 294
Observer Design Pattern

The [diagram on Wikipedia](https://en.wikipedia.org/wiki/Observer_pattern) uses a slightly different object modeling conventions, but uses the same general idea.
Yet another diagram from the GoF book...

“setState” refers to some setter method of the subject.
Observer/Observable Pattern - Why?

- **Simple way to decouple modules**
  - An observable doesn’t need to know much about its observers - As long as they implement some (usually simple) observer interface, they will get notified whenever something interesting happens.

- **Fundamental building block in event-driven architecture**
  - Common architecture where different components interact by raising (and listening to) events.
  - Extremely common architecture for modern distributed systems.
  - Another common example: Graphical User Interface, where a user’s action (such as a mouse-click) raises an event, which triggers various listeners. This is a standard way of decoupling GUI (presentation layer) from business logic.
Observer Design Pattern

Let’s go through an example.
First implementation
Observer in Java

- Issues with our first implementation:
  - What if Stock was already extending some class other than Observable?
    - Can only extend one class
  - The Observer interface requires us to cast the (Observable) argument (as Stock)
    - Another thing that can go wrong at runtime
    - Due to backwards compatibility
Roll own Observer/Observable
Observer in Java

- Issues with our **second implementation**:
  - We need to write more code (essentially, implementing what’s already implemented in Java’s built-in `Observable` class)
    - If you look at `java.util.Observable`, you’ll see that it’s not trivial code.
    - Perhaps more complicated than you expected?
  - More code ⇒ More work
  - More code ⇒ Potentially more bugs
Observer in Java

- An issue with observers:
  - Observables keep references to their observers ⇒ Observers don’t get garbage collected, until they are removed
    - Can lead to memory leaks
    - Exists in any language with automatic garbage collection
    - Known as the lapsed listener problem.
  - One solution is: Weak references
Back To Our Example

● Let’s introduce a new “requirement” ...

● When the application observes a change in a stock, it would like to get the old price, as well as the new price.
New Requirement

- We can handle with it in many ways:
  - Make the application store every price update in some sort of a database that it can query.
    - Very heavy solution.
  - Make the application store just the last price update it has seen for each stock.
    - Better, but incohesive - The application has a new, unrelated responsibility
  - Create an adapter ...
The Adapter Design Pattern

- Simple and intuitive
  - Object A outputs data in format 1
  - Object B expects input in format 2
  - Adapter connects between Object A and Object B
    - Takes input in format 1
    - Generates output in format 2
- The concept should seem familiar
Implementing The Adapter Pattern

Structure
A class adapter uses multiple inheritance to adapt one interface to another:

An object adapter relies on object composition:

Inheritance vs. Composition
The Adapter Design Pattern

● Cleans up your code

● Helps decouple modules and separate concerns
  ○ No need to worry about adapting to other modules’ expectations

● Sometimes it’s the only solution
  ○ Ex: Adapting between components you cannot change (You don’t have the source code)
  ○ Especially when combining frameworks
  ○ Original *GoF* example combined a font framework (for text) with a drawing program framework.
Adapter to combine Frameworks

two approaches correspond to the class and object versions of the Adapter pattern.

We call TextShape an adapter.

TextView part of font manager framework

TextShape adapts TextView to drawing editor framework
Observers & Adapters, Summary

- Extremely common patterns
- Tools for “gluing stuff together”
- Help us obtain
  - Cleaner code
  - Cohesive classes
  - Decoupled system
Test Doubles - Design Patterns For Testing

- Various design patterns for objects used for testing
- The idea is simple - During testing, replace an object with a test double
  - An object that “looks the same” (i.e. implements the same interface(s) )
  - But gives us more inspection/monitoring capabilities
Test Doubles

- Test double is a general name for various patterns:
  - Dummy
    - No implementation, usually used just to fill parameter requirements
  - Fake
    - Implementation not suitable for production use (e.g. in-memory test database)
  - Stub
    - Somewhat similar to Fake, but only behaviour needed for testing is implemented
  - Spy
    - A stub that also keeps information about how it was called
  - Mock
    - A spy that expects to be used a certain way
Test Doubles

- There is a very interesting article about **mocks vs. stubs**
  - With good arguments for and against each approach
- More than a subtle difference in implementation details, it’s a different testing philosophy!
  - In the “classic approach” (using stubs), we validate the state of an object. That is, we call method X on object A, and then inspect A to make sure it looks as we expect.
  - In the “mockist approach” (using mocks), we validate the behaviour. That is, we call method X on object A, and verify that A made the expected calls (i.e. sent the expected messages) to all of its collaborators.
Another Code Example

- Let’s see another code example that applies the observer pattern.
  - It is “the classic” use case of the observer/observable pattern - Separating User Interface from “the engine” of your application
  - It also uses a test double to test the observable functionality (i.e. Verify that the observable is notifying all of its observers properly)
MVC

● Using observers and/or adapter to separate between UI and backend is a very common technique.

● It evolved into a pattern/paradigm, called MVC
  ○ MVC stands for Model, View, Controller.
  ○ Divide an application into three pieces:
    ■ Model - Our application’s representation of the real-world
    ■ View - What users interact with
    ■ Controller - Connects between the model and the view

● Let’s see an example of MVC