CSE 331
Software Design & Implementation

Kevin Zatloukal
Fall 2017
Events, Listeners, and Callbacks
(Based on slides by Mike Ernst, Dan Grossman, David Notkin, Hal Perkins, Zach Tatlock)
Reminders

• Quiz 5 due tonight

• HW8 & 9 both posted
  – start thinking about features you want for HW9
    can add them to your model now
  – model is usually much easier to test than view/controller
Review: Module Design

• Want to reduce coupling between modules
  – makes each difficult to build & reason about independently
  – makes bugs more likely
  – makes each module more difficult to change
  – (recall the properties of high quality code...)

• Want cohesion within a single module
  – lack of cohesion suggests it could be split

• Superclass and subclass are often tightly coupled
  – unseen dependencies such as patterns of “self calls”
  – EJ: prefer composition

• Today: design examples and patterns to improve designs
Design exercise #1

Write a typing-break reminder program

Offer the hard-working user occasional reminders of the perils of Repetitive Strain Injury, and encourage the user to take a break from typing.

Naive design:

– Make a method to display messages and offer exercises
  – Make a loop to call that method from time to time
public class TimeToStretch {

    public void run() {
        System.out.println("Hey, you! Stop typing!");
        suggestExercise();
    }

    public void suggestExercise() {
        ...
    }
}

TimeToStretch suggests exercises
Timer calls run() periodically

```java
public class Timer {
    private TimeToStretch tts = new TimeToStretch();
    public void start() {
        while (true) {
            ...
            if (enoughTimeHasPassed) {
                tts.run();
            }
            ...
        }
    }
}
```
Main class puts it together

class Main {
    public static void main(String[] args) {
        Timer t = new Timer();
        t.start();
    }
}

This program, as designed, will work...
    But we can do better
An arrow in a module dependency diagram (MDD) indicates “depends on” or “knows about”

– simplistically: “any name mentioned in the source code”

What’s wrong with this diagram?

– does Timer really need to depend on TimeToStretch?

– is Timer re-usable in a new context?
Decoupling

**Timer** needs to call the **run** method

- **Timer** does *not* need to know what the **run** method does

Weaken the dependency of **Timer** on **TimeToStretch**

- introduce a *weak* specification for what **Timer** needs

```java
public interface TimerTask {
    public void run();
}
```

**Timer** only needs to know that something (e.g., **TimeToStretch**) meets the **TimerTask** specification
public class TimeToStretch implements TimerTask {
    public void run() {
        System.out.println("Stop typing!");
        suggestExercise();
    }

    public void suggestExercise() {
        ...
    }
}
Timer (version 2)

```java
public class Timer {
    private TimerTask task;
    public Timer(TimerTask task) {
        this.task = task;
    }
    public void start() {
        while (true) {
            ... 
            task.run();
        }
    }
}
```

Main creates a `TimeToStretch` object and passes it to `Timer`:
```java
Timer t = new Timer(new TimeToStretch());
t.start();
```
Module dependency diagram (version 2)

- Timer depends on TimerTask, not TimeToStretch
  - unaffected by implementation details of TimeToStretch
  - now Timer is much easier to reuse
  - Main depends on the constructor for TimeToStretch
- Main still depends on Timer (is this necessary?)
The callback design pattern

An alternative: use a callback to *invert the dependency*

**TimeToStretch** creates a **Timer**, and passes in a reference to *itself* so the **Timer** can *call it back*

- this is a *callback*
- call from module to a client that it notifies about some condition

The callback *inverts a dependency*

- inverted dependency: **TimeToStretch** depends on **Timer** (not vice versa)
  - less obvious coding style, but more “natural” dependency
- side benefit: **Main** does not depend on **Timer**
Callbacks

Callback: “code” provided by client to be used by library
• in Java, pass an object with the “code” in a method

Synchronous callbacks:
• useful when library needs the callback result immediately
• examples: HashMap calls its client’s hashCode, equals

Asynchronous callbacks:
• register to indicate interest and where to call back
• examples: GUI events, timers
• useful when the callback should be performed later (when some interesting event occurs)
public class TimeToStretch extends TimerTask {
    private Timer timer;
    public TimeToStretch() {
        timer = new Timer(this);
    }
    public void start() {
        timer.start();
    }
    public void run() {
        System.out.println("Stop typing!");
        suggestExercise();
    }
    ...
}
Main (version 3)

```java
TimeToStretch tts = new TimeToStretch();
tts.start();
```

- uses a callback in `TimeToStretch` to invert a dependency
- this MDD shows the inversion of the dependency between `Timer` and `TimeToStretch` (compare to version 1)
Decoupling and design

- Good design has dependences (coupling) only where sensible

- While you design (*before* you code), examine dependences
  - don’t introduce unnecessary coupling!

- Coupling is an easy temptation if you code first
  - suppose a method needs information from another object:
  - if you hack in a way to get it:
    - will damage the code’s modularity and reusability
    - more complex code is harder to understand
  - (coupling is the “friction” of building large software)
Design exercise #2

A program to display information about stocks
  – stock tickers
  – spreadsheets
  – graphs

Naive design:
  – make a class to represent stock information
  – that class updates all views of that information (tickers, graphs, etc.) when it changes
Main class gathers information and stores in **Stocks**

**Stocks** class updates viewers when necessary

Problem: To add/change a viewer, must change **Stocks**

- problem **only** if we want to allow others to add new viewers

Better: insulate **Stocks** from the vagaries of the viewers
Weaken the coupling

What should Stocks class know about viewers?

- only needs an update method to call with changed data
- old way:

```java
void updateViewers() {
    ticker.update(newPrice);
    spreadsheet.update(newPrice);
    graph.update(newPrice);
    // Edit this method to
    // add a new viewer. 😊
}
```
Weaken the coupling

What should Stocks class know about viewers?

- only needs an `update` method to call with changed data
- new way: The “observer pattern”

```java
interface PriceObserver {
    void update(PriceInfo pi);
}

class Stocks {
    private List<PriceObserver> observers;
    void addObserver(PriceObserver pi) {
        observers.add(pi);
    }
    void notifyObserver(PriceInfo i) {
        for (PriceObserver obs : observers)
            obs.update(i);
    }
    ...
}
```

Register a callback

Do the callbacks
The observer pattern

- **Stocks** not responsible for viewer creation
- **Main** passes viewers to **Stocks** as **observers**
- **Stocks** keeps list of **PriceObservers**, notifies them of changes

- **Issue:** `update` method must pass enough information to (unknown) viewers
A different design: pull versus push

- The Observer pattern implements *push* functionality.
- A *pull* model: give viewers access to *Stocks*, let them extract the data they need.

"Push" versus "pull" efficiency can depend on frequency of operations. (Also possible to use both patterns simultaneously.)
Another example of Observer pattern

// Represents a sign-up sheet of students
public class SignupSheet extends Observable {
    private List<String> students
        = new ArrayList<String>();
    public void addStudent(String student) {
        students.add(student);
        setChanged();
        notifyObservers();
    }
    public int size() {
        return students.size();
    }
}

SignupSheet inherits many methods including:
    void addObserver(Observer o)
    protected void setChanged()
    void notifyObservers()
public class SignupObserver implements Observer {
    // called whenever observed object changes
    // and observers are notified
    public void update(Observable o, Object arg) {
        System.out.println("Signup count: "+((SignupSheet)o).size());
    }
}
Registering an observer

```
SignupSheet s = new SignupSheet();
s.addStudent("billg");
// nothing visible happens
s.addObserver(new SignupObserver());
s.addStudent("torvalds");
// now text appears: "Signup count: 2"
```

Java's “Listeners” (particularly in GUI classes) are examples of the Observer pattern

(Feel free to use the Java observer classes in your designs – if they are a good fit – but you don’t have to use them)
User interfaces: appearance vs. content

It is easy to tangle up *appearance* and *content*

- especially when supporting direct manipulation
  (e.g., dragging line endpoints in a drawing program)
- example: program state stored in widgets in dialog boxes

Neither can be understood easily or changed easily

This destroys modularity and reusability

- over time, it leads to bizarre hacks and huge complexity

Callbacks, listeners, and other patterns can help
Advice

• Worry about dependencies
  – they make code hard to change

• But also worry about simplicity
  – sometimes the cure is worse than the disease
  – don’t introduce lots of new concepts and abstraction in order to fix what is not really a problem
  – Example: what if ticker, spreadsheet, and graph are the only observers we ever need?