CSE 331
Software Design & Implementation

Hal Perkins

Fall 2016

Events, Listeners, and Callbacks

The limits of scaling

What prevents us from building huge, intricate structures that work perfectly and indefinitely?

- No friction
- No gravity
- No wear-and-tear

… it's the difficulty of *understanding* them

So we split designs into sensible parts and reduce interaction among the parts

- More *cohesion* within parts
- Less *coupling* across parts

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

(Let's ignore multithreaded solutions for this discussion)

`TimeToStretch` suggests exercises

```
public class TimeToStretch {
    public void run() {
        System.out.println("Stop typing!");
        suggestExercise();
    }
    public void suggestExercise() {
        …
    }
}
```

`Timer` calls `run()` periodically

```
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();
  }
}
```

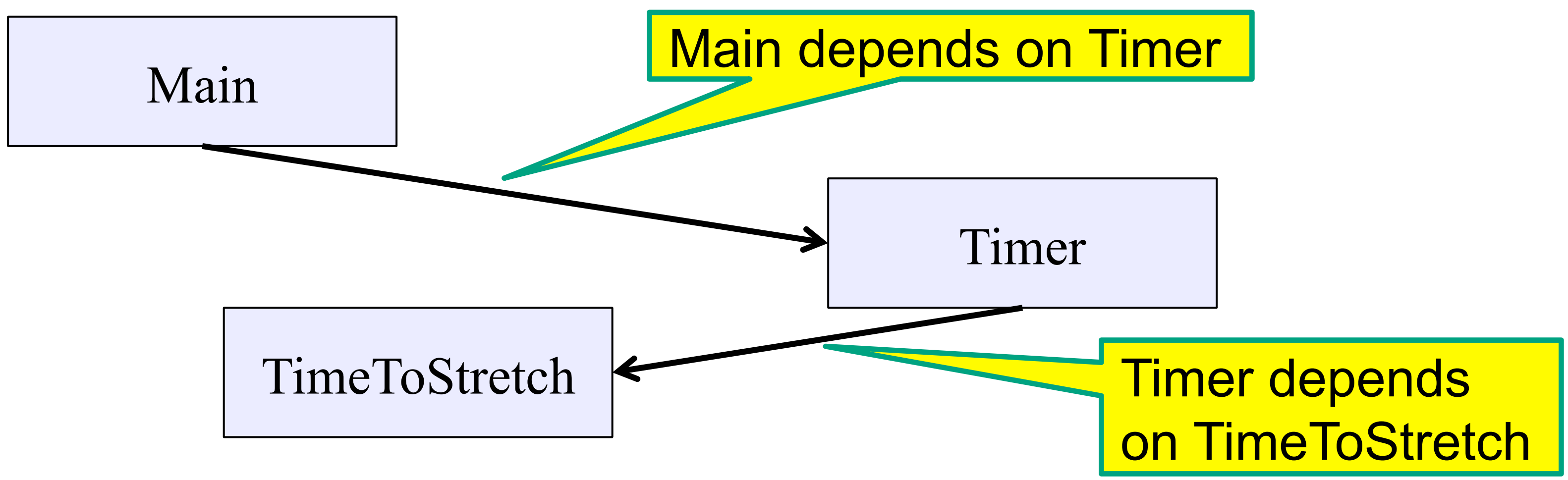
This program, as designed, will work...

- But we can do better

Module dependency diagram (MDD)

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 weaker specification, in the form of an interface or abstract class

```
public abstract class TimerTask {
    public abstract void run();
}
```

`Timer` only needs to know that something (e.g., **`TimeToStretch`**) meets the **`TimerTask`** specification

TimeToStretch (version 2)

```
public class TimeToStretch extends TimerTask {
    public void run() {
        System.out.println("Stop typing!");
        suggestExercise();
    }

    public void suggestExercise() {
         ...
    }
}
```

Timer (version 2)

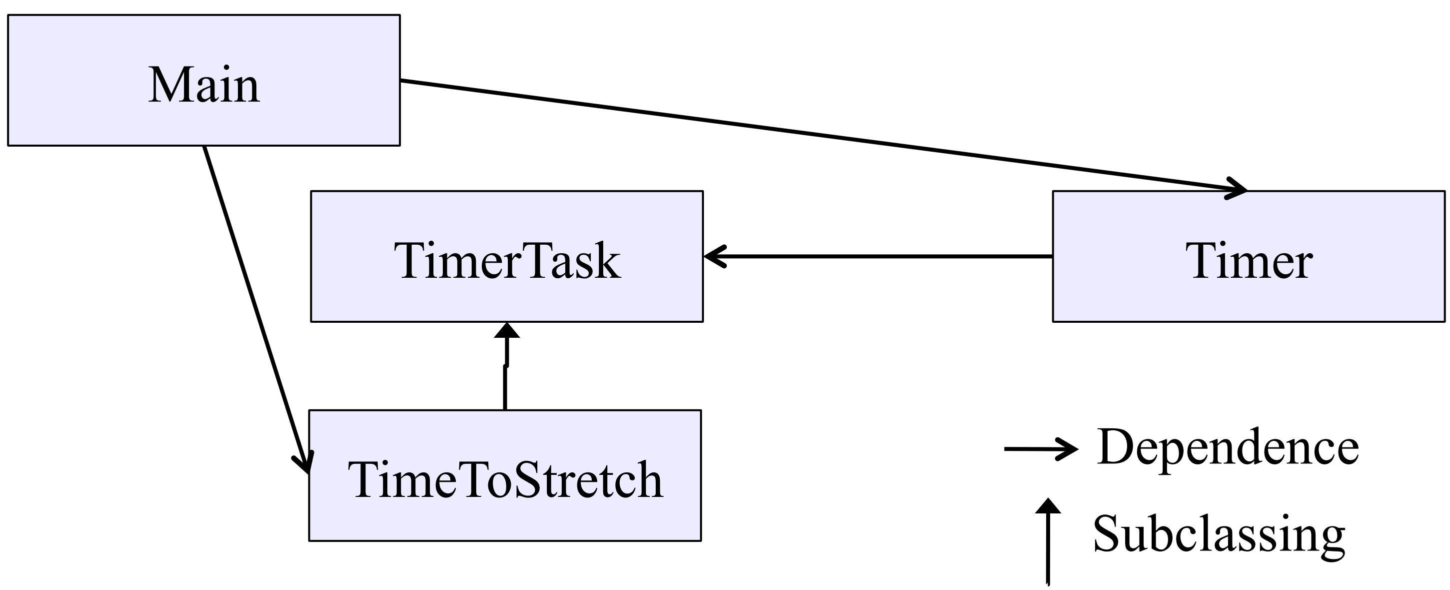
```
public class Timer {
  private TimerTask task;
  public Timer(TimerTask task) {
     this.task = task;
  }
  public void start() {
    while (true) {
      ...
      task.run();
    }
  }
}
```

Main creates a **`TimeToStretch`** object and passes it to **`Timer`**:

```
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* – a method call from a 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:

- Examples: **`HashMap`** calls its client's **`hashCode`**, **`equals`**
- Useful when library needs the callback result immediately

Asynchronous callbacks:

- Examples: GUI listeners
- *Register* to indicate interest and where to call back
- Useful when the callback should be performed later, when some interesting event occurs

TimeToStretch (version 3)

```
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();
    }
    ...
}
```

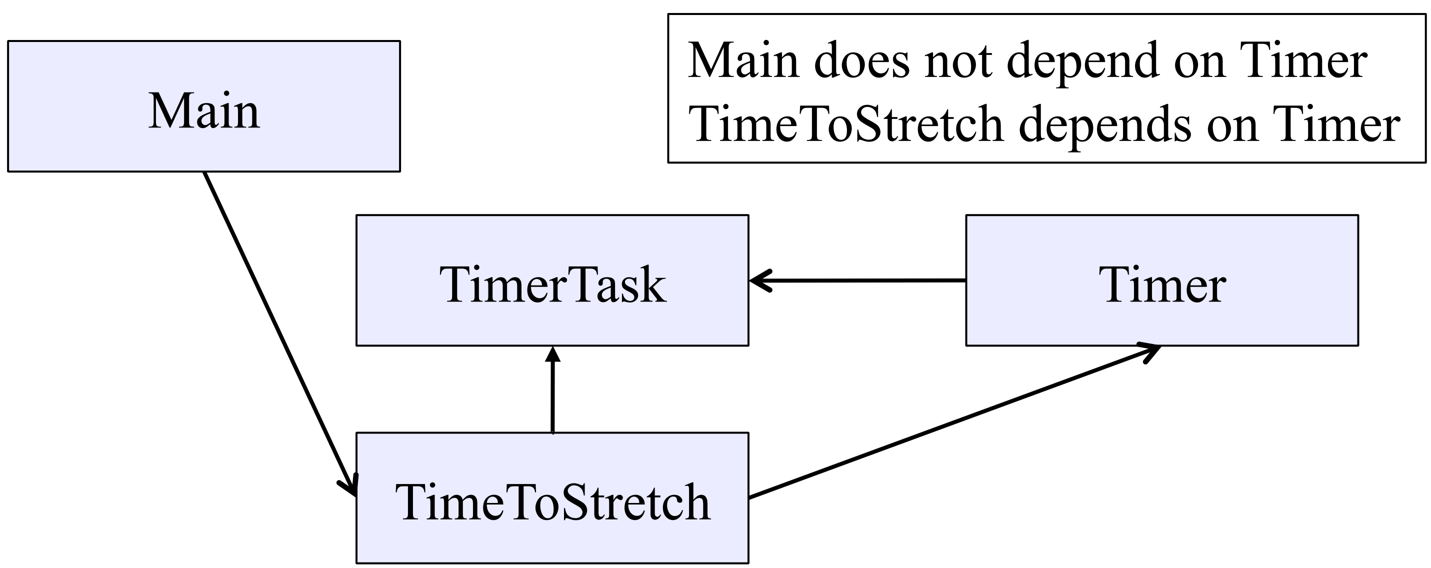
Register interest with the timer

Callback entry point

Main (version 3)

```
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

- A good design has dependences (coupling) only where it makes sense
- 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:
    - The hack might be easy to write
    - It will damage the code's modularity and reusability
    - More complex code is harder to understand

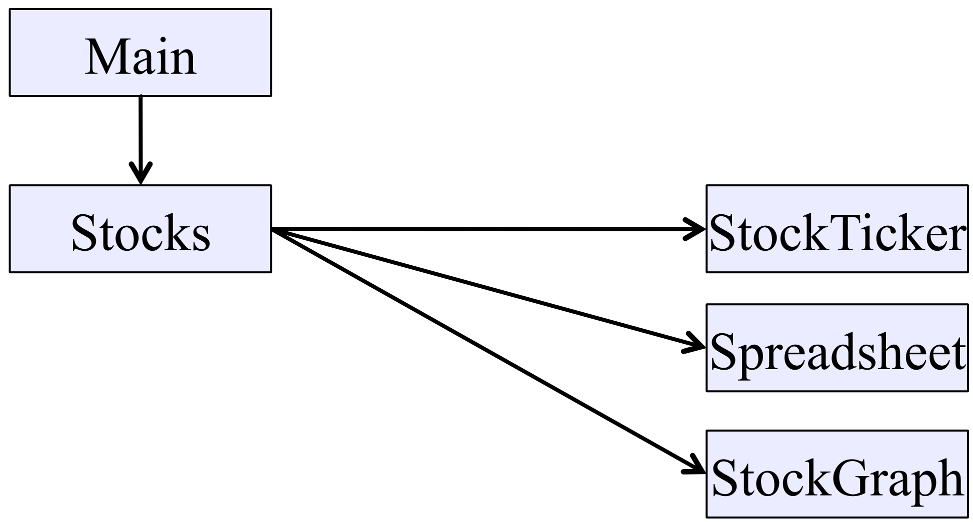
Design exercise #2

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

Module dependency diagram

- Main class gathers information and stores in **`Stocks`**
- **`Stocks`** class updates viewers when necessary



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

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:

```
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”

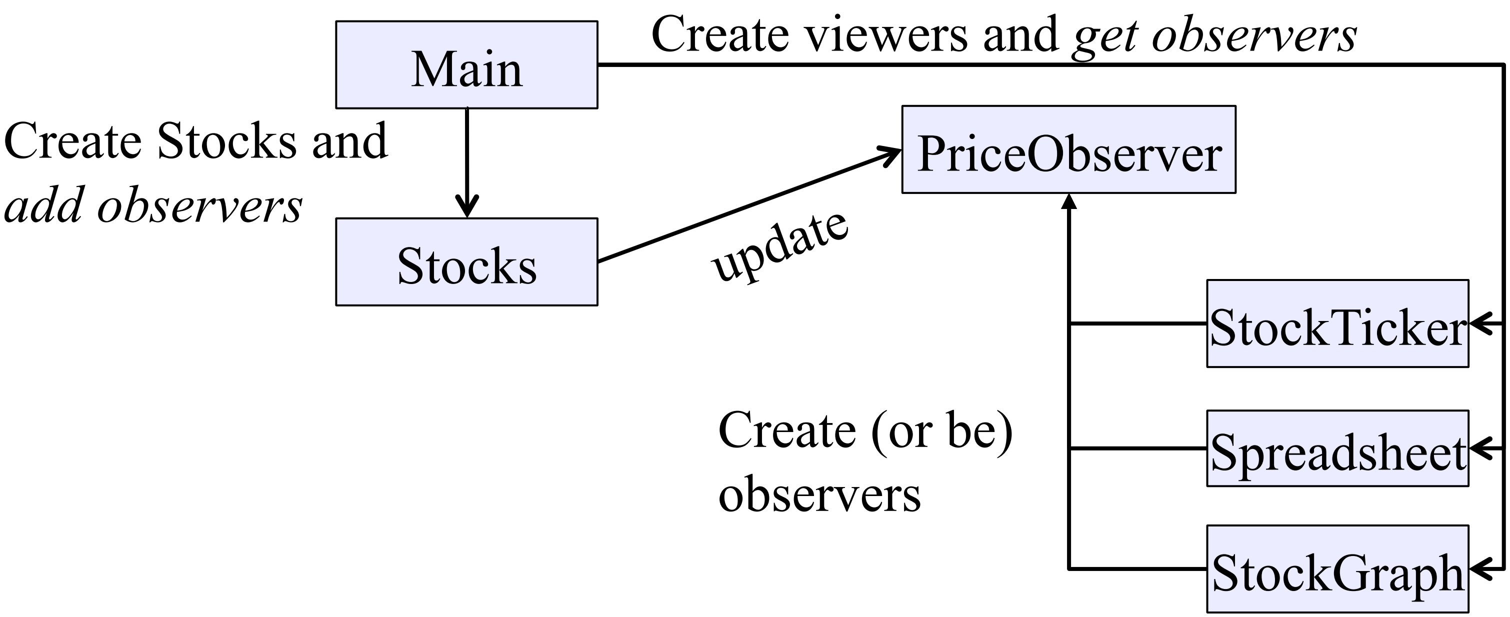
```
interface PriceObserver {
  void update(PriceInfo pi);
}

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

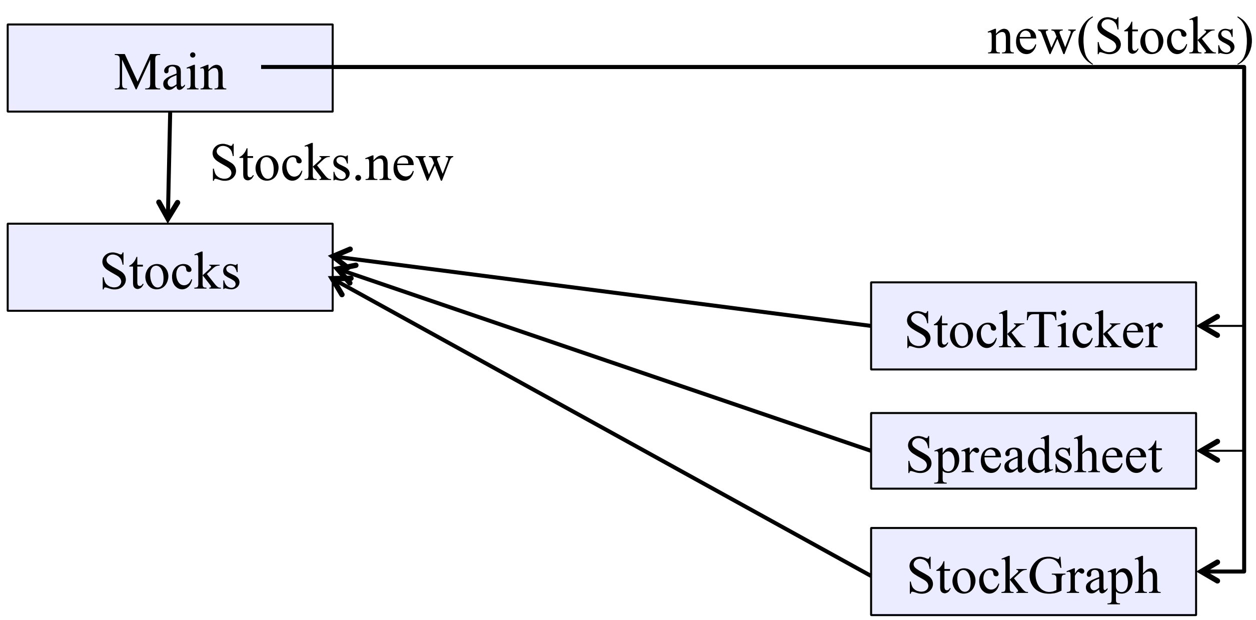
- **`Stocks`** not responsible for viewer creation
- **`Main`** passes viewers to **`Stocks`** as *observers*
- **`Stocks`** keeps list of **`PriceObservers`**, notifies them of changes



- Issue: **`update`** method must pass enough information to (unknown) viewers

A different design: pull versus push

- The Observer pattern (last slide) implements *push* functionality
- Alternative: 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();
  }
  …
}
```

Part of the JDK

`SignupSheet` *inherits* many methods including:

```
void addObserver(Observer o)
protected void setChanged()
void notifyObservers()
```

An Observer

Part of the JDK

```
public class SignupObserver implements Observer {
  // called whenever observed object cha
  // and observers are notified
  public void update(Observable o, Object arg) {
    System.out.println("Signup count: "
                       + ((SignupSheet)o).size());
  }
}
```

Not relevant to us

cast because **`Observable`** is not generic ☹

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*

- Particularly when supporting direct manipulation (e.g., dragging line endpoints in a drawing program)
- Another 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
- Code must be discarded

Callbacks, listeners, and other patterns can help