Module dependences and decoupling

(Events, listeners, callbacks)

CSE 331
University of Washington

Michael Ernst

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

… the difficulty of managing complexity (e.g., understanding them)

Solution: Modularity, and minimize interactions

Interactions cause complexity

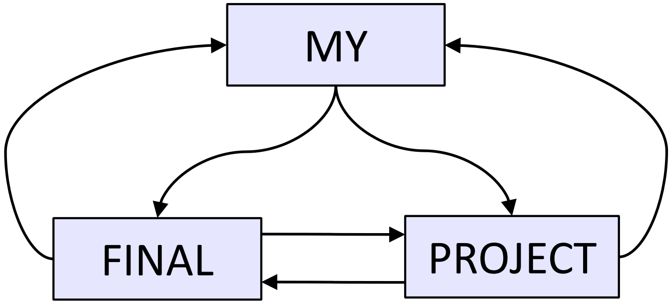
To simplify, split design into parts that don't interact much

Coupling: amount of interaction between parts

Cohesion: similarity within a part

MY
FINAL
PROJECT

An application

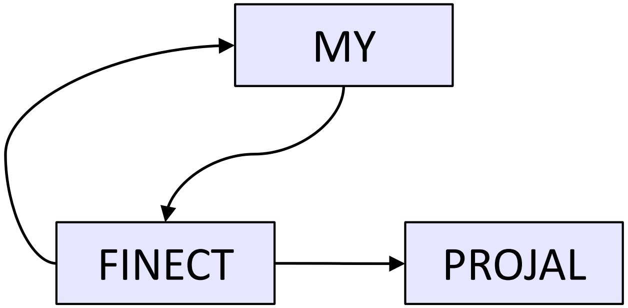


MY

FINAL

PROJECT

A poor decomposition (parts strongly coupled)



MY

FINECT

PROJAL

*Not the obvious decomposition, but **better** (parts weakly coupled)*

Design exercise #1

Write a typing break reminder program

Remind the user about Repetitive Strain Injury, and encourage the user to take a break from typing

Naive design:

- Main program makes a timer
- Timer loop performs action periodically
- Action = display messages and offer exercises

(Let's ignore multi-threaded 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();
  }
}
```

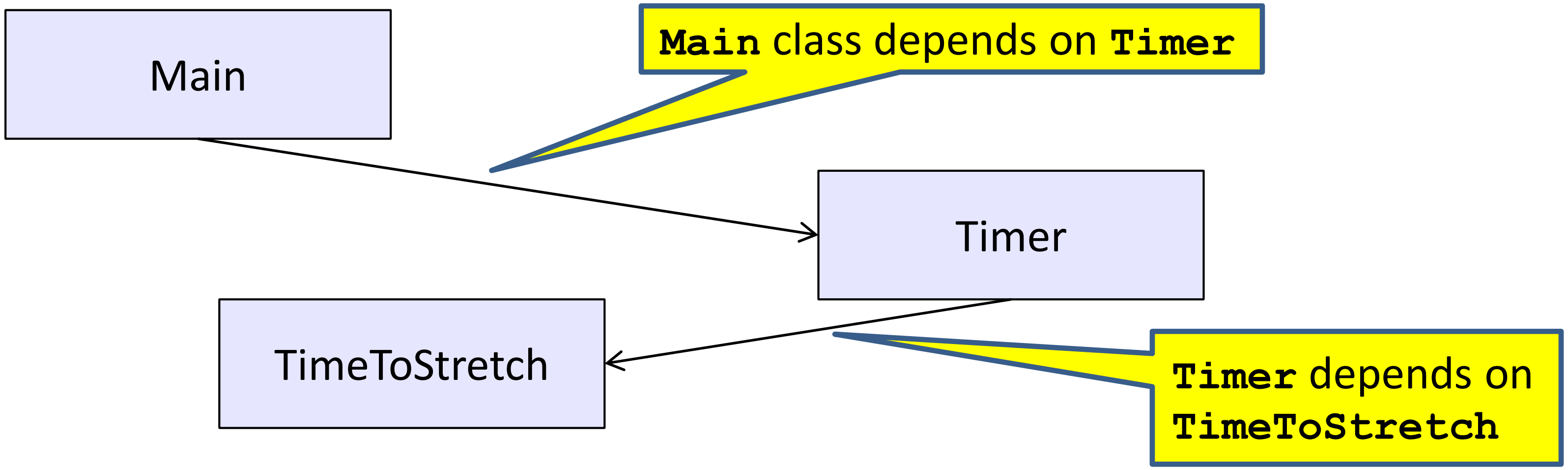
This program, as designed, will work...

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



`Main` class depends on **`Timer`**

`Timer` depends on **`TimeToStretch`**

What is wrong with this design?

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` uses **`TimerTask`**, works with anything that meets the **`TimerTask`** specification (e.g., **`TimeToStretch`**)

TimeToStretch (version 2)

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

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

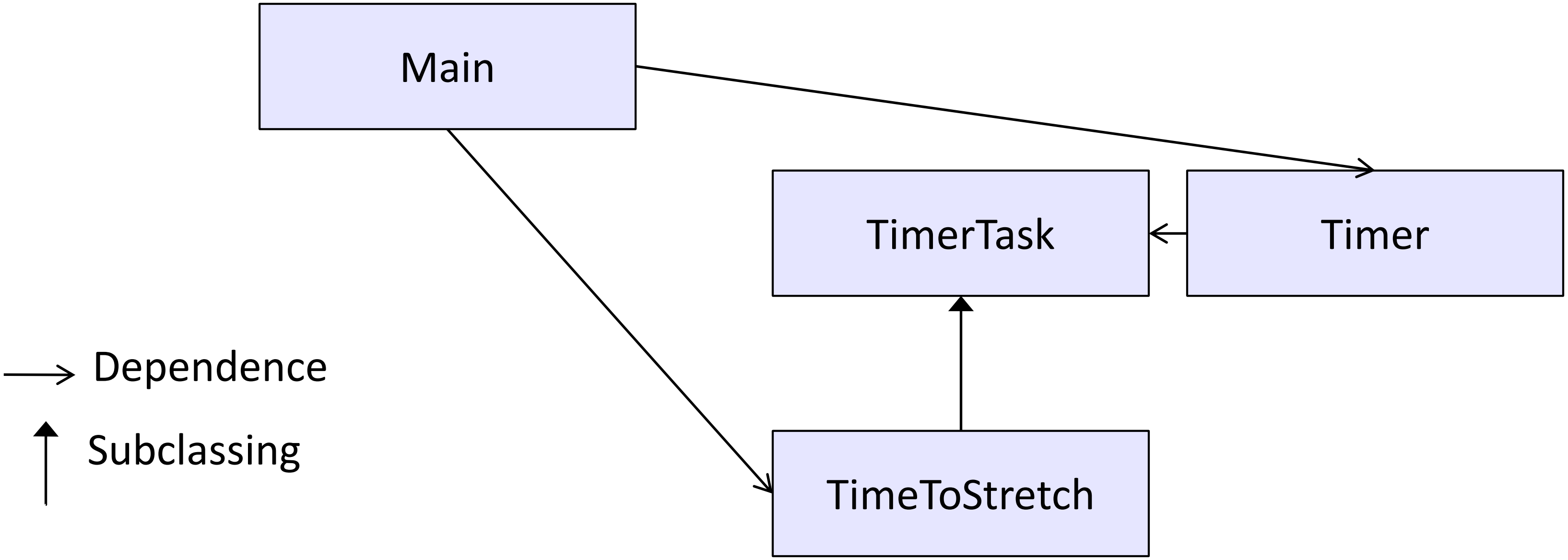
Timer (version 2)

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

`Main` creates the `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`**

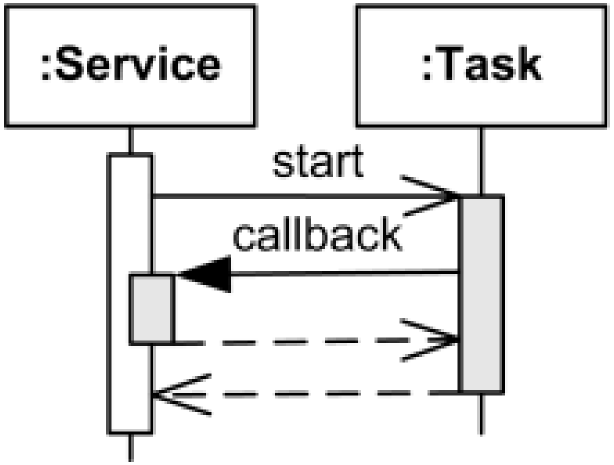
- **`Main`** still depends on **`Timer`**. Is this necessary?

The callback design pattern

- A computes
- A calls B
  - B computes
  - *Before* B completes, B calls A
    - A computes
    - A returns a value
  - B computes more
  - B returns
- A computes more

- Example: Factory object

- Advantage: B does not depend on A
  - B depends on some superclass of A



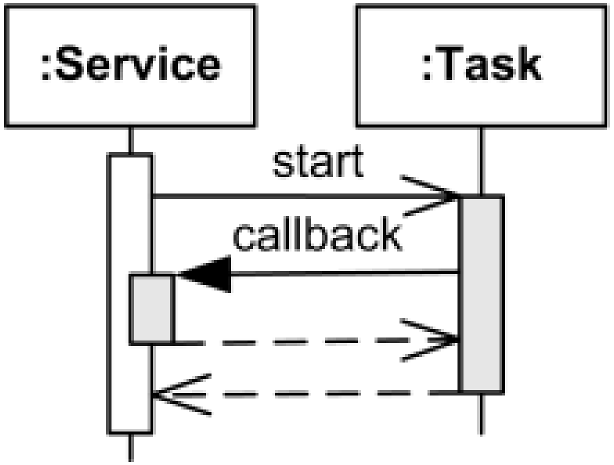
A synchronous callback.
Time increases downward.
Solid lines: calls
Dotted lines: returns

It's a callback whenever the stack contains:

my code
library code
my code

even if the two "my code" are not the same object or class

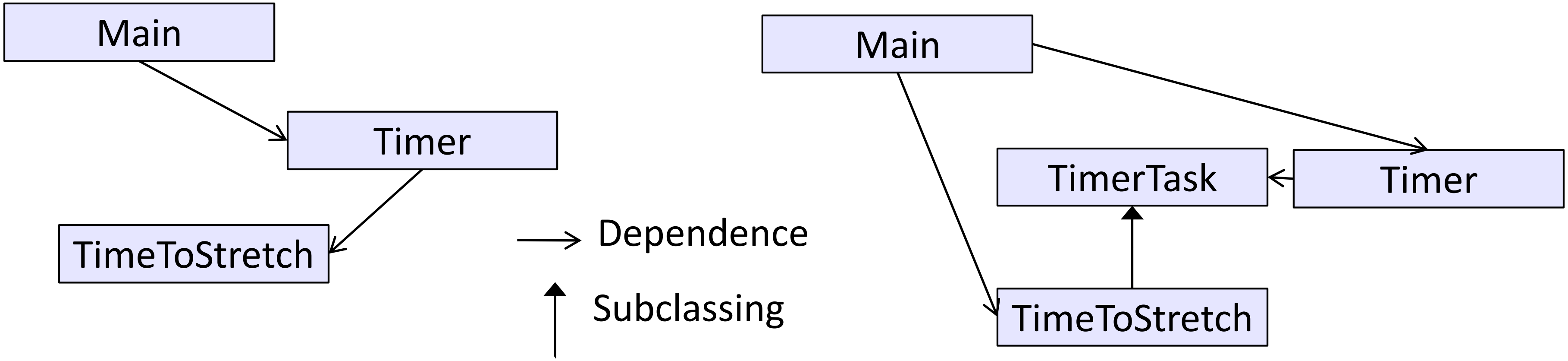
Examples of callbacks



A synchronous callback.
Time increases downward.
Solid lines: calls
Dotted lines: returns

- Synchronous callbacks:
  - Examples: **`HashMap`** calls its client's **`hashCode`**, **`equals`**
  - Useful when the callback result is needed immediately by the library
- 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

Use a callback to invert a dependency



Main

Timer

TimeToStretch

Dependence

Subclassing

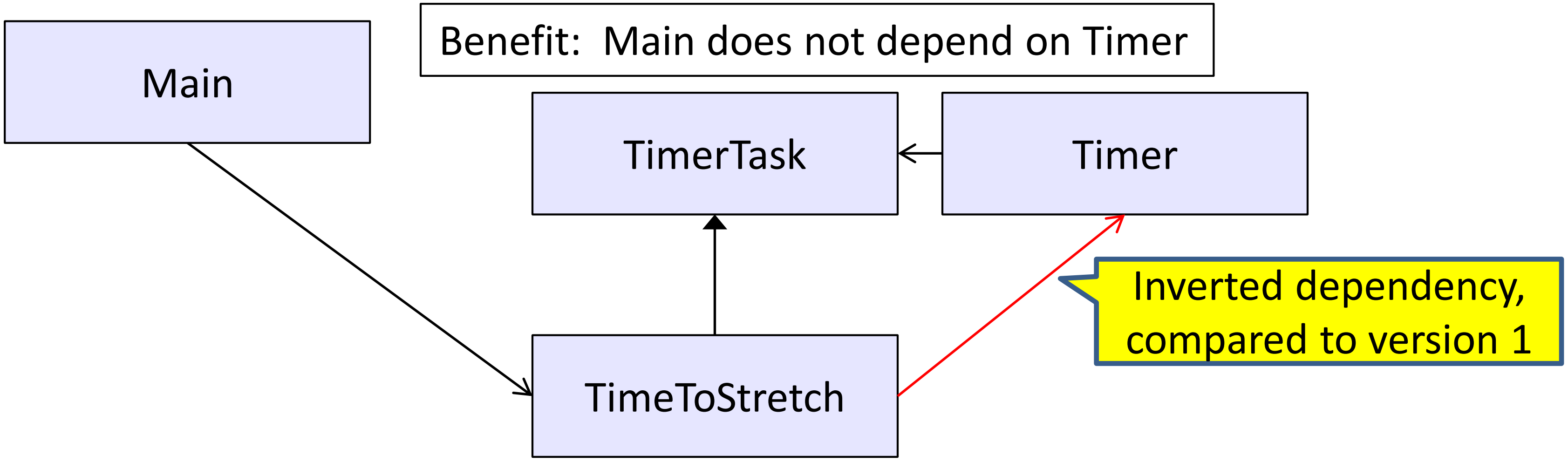
Main

TimerTask

Timer

TimeToStretch

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



Benefit: Main does not depend on Timer

Main

TimerTask

Timer

TimeToStretch

Inverted dependency, compared to version 1

TimeToStretch (version 3)

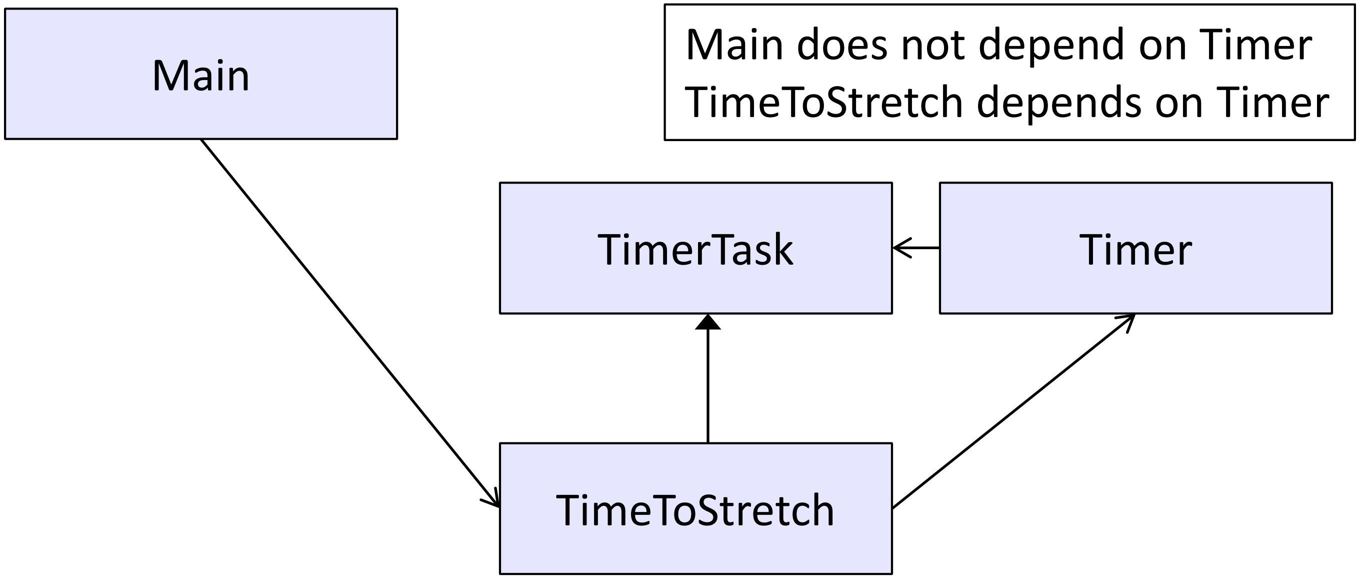
```
public class TimeToStretch extends TimerTask {
  private Timer timer;
  public TimeToStretch() {
    timer = new Timer(this);
  }
  public void start() {
    timer.start();
  }
  @Override
  public void run() {
    System.out.println("Stop typing!");
    suggestExercise();
  }
  ...
}
```

Register interest with the timer

Callback entry point

Main (version 3)

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



Main does not depend on Timer
TimeToStretch depends on Timer

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

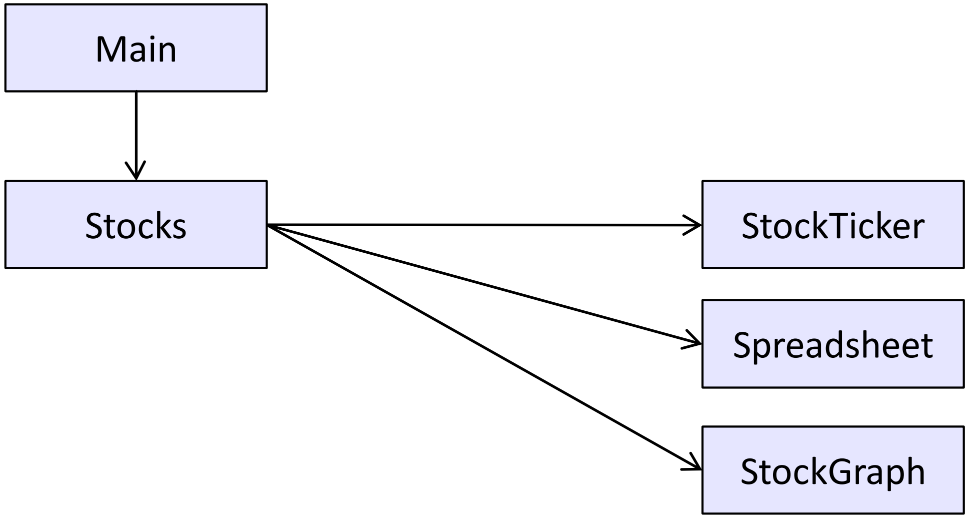
Design exercise #2

- 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`**

It is better to insulate **`Stocks`** from the vagaries of the viewers

Weaken the coupling

What should Stocks class know about viewers?

Stocks needs to call viewers’ update method when price changes

Old:

```
void updateViewers() {
    myTicker.update(newPrice);
    mySpreadsheet.update(newPrice);
    myGraph.update(newPrice);
    // Edit this method whenever
    // different viewers are desired. ☹
}
```

New (uses “observer pattern”):

```
class Stocks {
  List<PriceObserver> observers;
  void notifyObserver() {
    for (PriceObserver obs : observers) {
      obs.update(newPrice);
    }
  }
}
interface PriceObserver {
    void update(...);
}
```

Callback

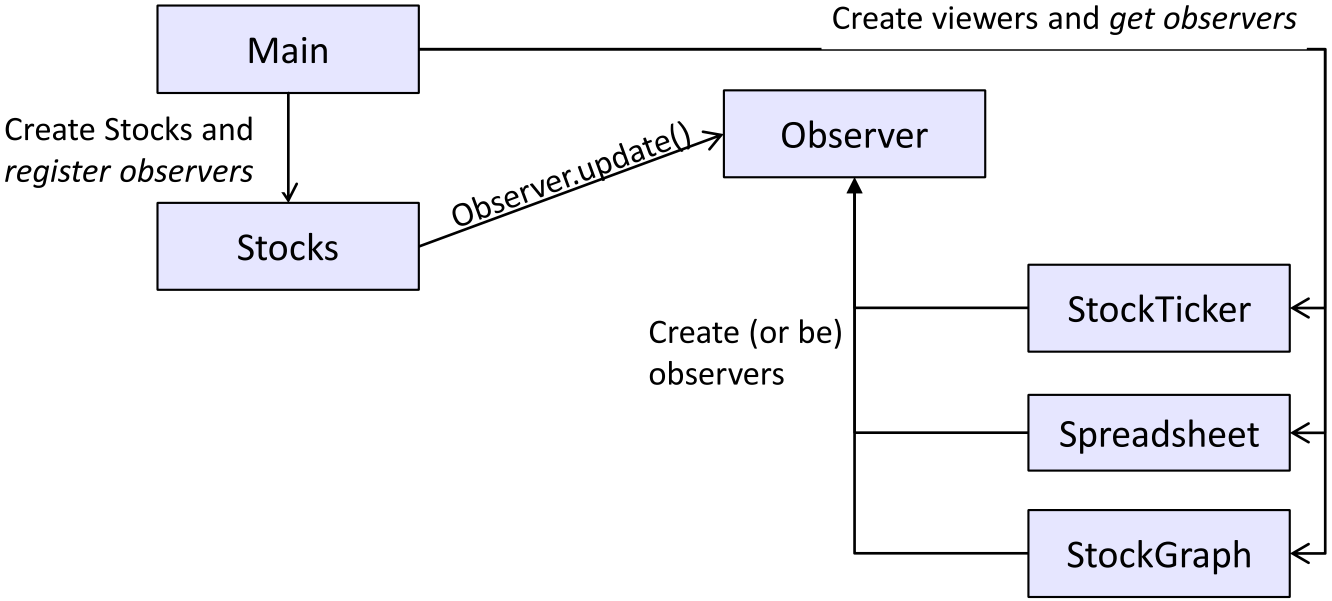
How are observers created and registered?

The observer pattern

Stocks are not responsible for viewer creation

Main passes viewers to Stocks as observers

Stocks keeps list of observers, notifies them of changes

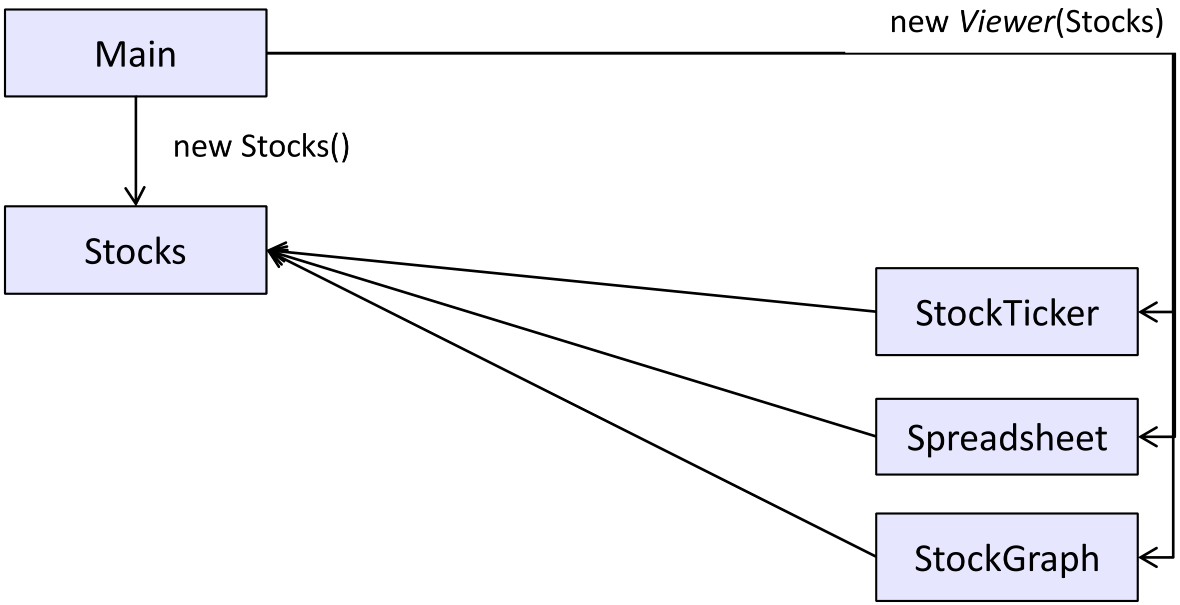


Issue: what info should **`update`** pass to unknown viewers?

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



The most efficient design (push or pull) depends on frequency of operations. (It's 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);
      notifyObservers();
   }
   public int size() {
      return students.size();
   }
}
```

Part of the JDK

Inherited from Observable class

An Observer

Part of the JDK

```
public class SignupObserver implements Observer {
  // callback that should be called whenever the
  // observed object changes, to notify this observer
  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 ☹

Using the 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

You may use Java observer classes in your designs, but you are not required to do so.

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

Shared constraints

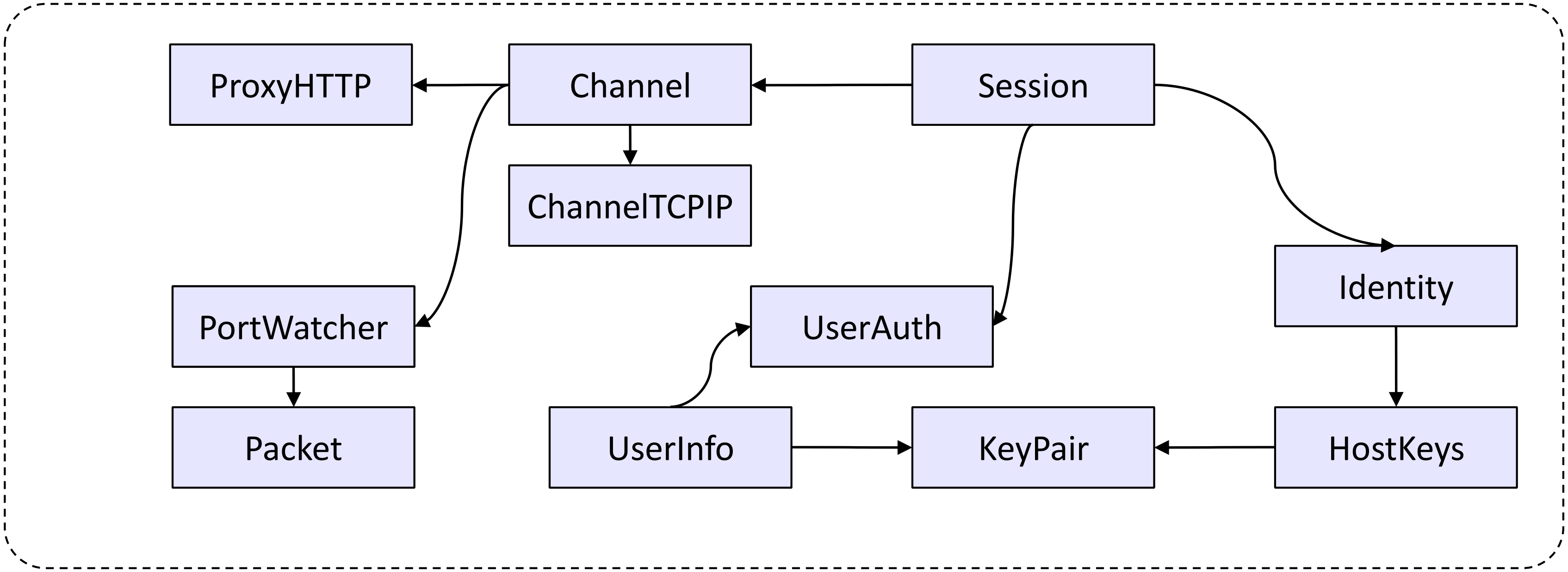
- Coupling can result from “shared constraints”, not just code dependencies
  - A module that writes a file and a module that reads the file depend on a common file format
    - Even if there is no dependency on each other's code
  - If one fails to write the correct format, the other will fail to read
- Shared constraints are easier to reason about if they are well encapsulated
  - A single module should contain and hide all information about the format

Facade

Want to perform secure file copies to a server

Given a general purpose library, powerful and complex

Good idea: build a facade – a new interface to that library that hides its (mostly irrelevant) complexity



Facade

If the library changes, you can update only SecureCopy