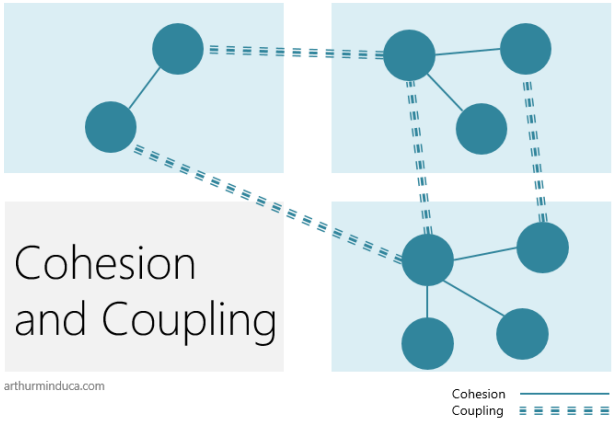
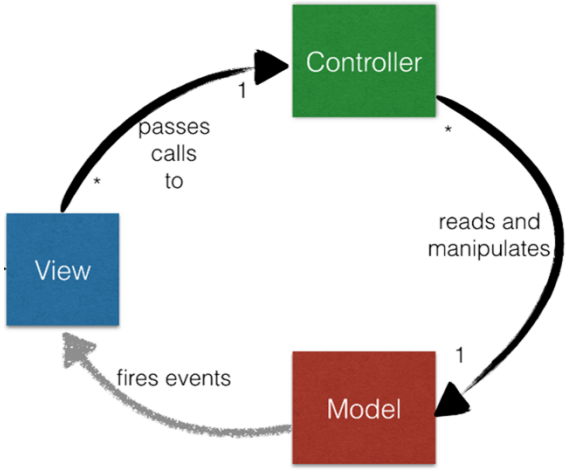
CS5010 – PDP

Introduction to Design





Maria Zontak

Slides inspired by slides of H.Perkins (UW, CS331)

and MIT 16.355 software engineering course

Disclaimer

- Design - a **creative** problem-solving.
- Bad news:

No recipe for doing it (maybe some in functional design)

- Good news:

Expertise and knowledge (from experience) of designers are determinant for success.

Design problem formulation

How to decompose system into parts, each with a lower complexity than the whole system, such that:

- Interaction between the parts is minimized
- Combination of these parts together solves the problem

- No universal way of doing this ☹ ☺

Rule of thumb:

- Do NOT think of system in terms of components that correspond to steps in processing.
- Do provide a set of modules that are useful for writing many programs.

Modules

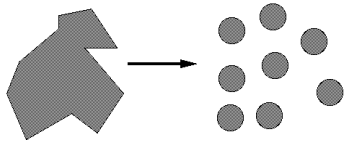
A *module* - relatively general term for a class or a type or any kind of design unit in software

A *modular design* focuses on what modules are defined, what their specifications are, how they relate to each other

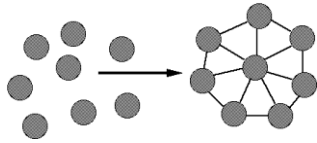
- Not the implementations of the modules
- Each module respects other modules' abstraction barriers!
- Each module should provide a single abstraction (an ADT)
- Each method should do one thing well

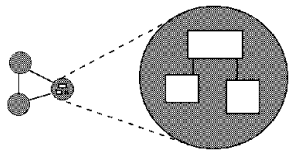
Ideals of modular software

Decomposition – design allows to break down a problem into modules to reduce complexity and allow teamwork

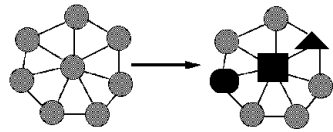


Composability – “Having divided to conquer, we must reunite to rule [M. Jackson].” **Ideally in various way!**
Necessary condition:
Self-contain (modular) – one module can be developed and examined in isolation

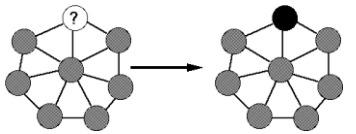




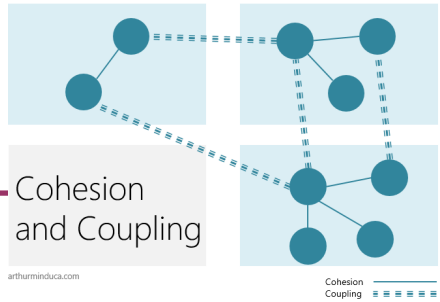
Continuity/Locality – a small change in the requirements should affect ideally ONE or a small number of modules



Isolation – an error in one module should be as contained as possible



Coupling and Cohesion



In your object oriented design
should the following properties be minimized/maximized?

- **Cohesion**
  - How well a module encapsulates a single notion/responsibility
  - Degree to which the elements of a module belong together
- **Coupling**
  - The degree to which a module interacts with or depends on another modules

OO design should be divided into modules/classes such that
Coupling between modules/classes is **minimized**
Cohesion within modules/classes is **maximized**

Let's put that into practice…

"Pay Attention" program

Provide the student in PDP course occasional reminders on his/her laptop screen to take a break from any unrelated activity, and encourage him/her to concentrate on the ongoing lecture…

Any suggestions for a design?

1. A class that is responsible to display a message "Pay Attention to the Lecture"
2. A class that is responsible to call that method from time to time

`Reminder`

`Timer`

“Pay Attention” Program - Naïve Design

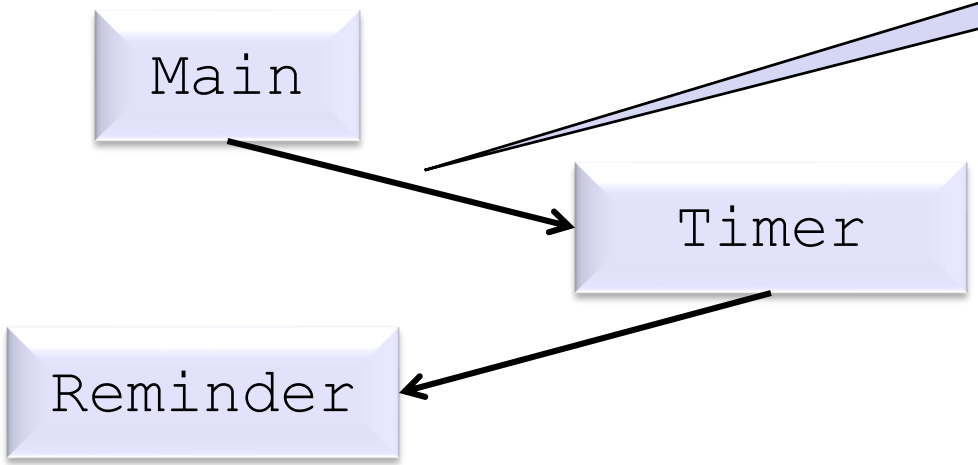
```
public class Reminder {
    public void display() {
      System.out.println(“Please Pay Attention to the Lecture!");
    }
}
```

```
public class Timer {
    private Reminder reminder = new Reminder();
    public void start() {
      while (true) {
          ...
          if (enoughTimeHasPassed) {
              reminder.display();
          }
          ...
      }
    }
}
```

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

"Pay Attention" Program - Naïve Design

Class/module dependency diagram:



"Pay Attention" Program - Naïve Design

```
public class Reminder {
    public void display() {
      System.out.println("Please Pay Attention to the Lecture!");
    }
}
```

```
public class Timer {
    private Reminder reminder = new Reminder();
    public void start() {
      while (true) {
          ...
          if (enoughTimeHasPassed) {
              reminder.display();
          }
          ...
      }
    }
}
```

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

Can we improve design/reduce dependencies?

- Is **`Timer`** REUSABLE (for another application)?
- Can **`Timer`** and **`Reminder`** be DECOUPLED?

Decoupling

Observation:

`Timer` needs to call the **`display`** method

BUT **`Timer`** does NOT need to know what the **`display`** does

- To decouple **`Timer`** and **`Reminder`** specify their relation via

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

- **`Timer`** will work with any class (including **`Reminder`**) if this class meets the **`TimerTask`** specification

“Pay Attention” Program - Improved Design

```
public class Reminder implements TimerTask {
    public void run() { display();}
    private void display() {
      System.out.println(“Please Pay Attention to the Lecture!");
    }
}
```

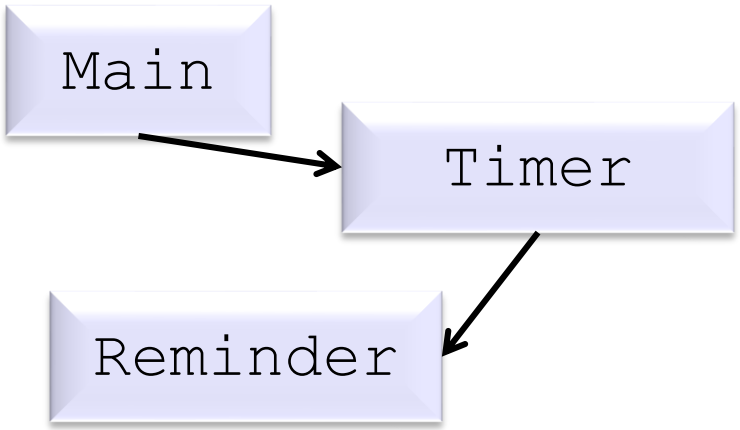
```
public class Timer {
    private TimerTask curTask;
    public Timer(TimerTask newTask) {
       curTask = newTask;
    }
    public void start()
      while (true) {
          ...
          if (enoughTimeHasPassed) {
          curTask.run();
          }
```

In **Main** :

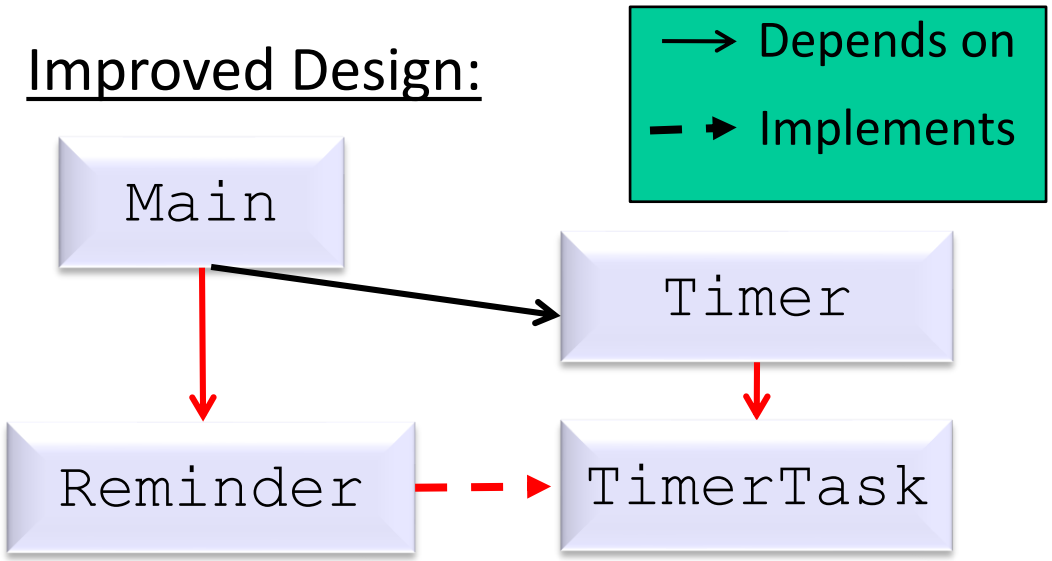
```
…
Timer timer = new Timer(new Reminder());
timer.run;
```

Let's compare both designs

Naïve Design:



Improved Design:



We achieved:

- **`Timer`** depends on **`TimerTask`**, NOT on **`Reminder`**

→ Unaffected by implementation details of **`Reminder`**

→ **`Timer`** can be reused

- **`Main`** depends on the constructor of **`Reminder`**

  and still depends on **`Timer`** (is this necessary?)

Alternative - using Callback Design Pattern

In real life, when you set reminder in your smartphone/calendar, will time depend on the reminder or reminder on the time?

→ We need to invert a dependency, such that **`Reminder`** depends on **`Timer`** (not vice versa)

[Less obvious coding style, but more "natural" dependency]

How?

CALLBACK: A method call from a module to a client, to notify it about some condition

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

- We have achieved our goal: **`Main`** does not depend on **`Timer`**

Callbacks

"Code" provided by a client to be used by a library

Synchronous Callbacks:

- Useful when library needs the callback result immediately, in a sequential order
- For examples: **`HashMap/HashSet`** calls its client's **`hashCode()`**, **`equals()`**

Asynchronous Callbacks:

- Useful when the callback should be performed later, upon occurrence of a relevant event
- Examples: GUI listeners

“Pay Attention” Program - Improved Design

```
public class Reminder implements TimerTask {
    public void run() { display();}
    private void display() {
      System.out.println(“Please Pay Attention to the Lecture!");
    }
}
```

```
public class Timer {
    private TimerTask curTask;
    public Timer(TimerTask newTask) {
       curTask = newTask;
    }
    public void start()
      while (true) {
          ...
          if (enoughTimeHasPassed) {
          curTask.run();
          }
```

In **Main** :

```
…
Timer timer = new Timer(new Reminder());
timer.run();
```

“Pay Attention” Program – using Callback

```
public class Reminder implements TimerTask {
    Timer timer;
    public Reminder () {
        timer = new Timer(this);
    }
    public void setOn() {
        timer.start();
    }
    public void run() { display();}
    private void display() {
      System.out.println(“Please Pay Attention to the Lecture!");
    }
}
```

Ask `timer` to call this `Reminder` back

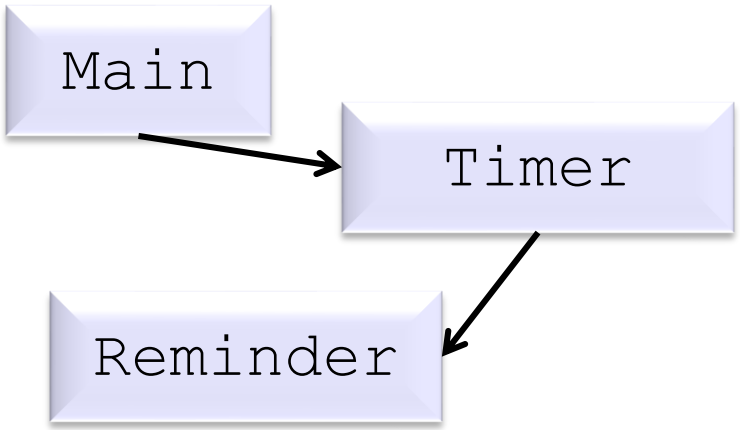
Callback entry point

In **Main** :

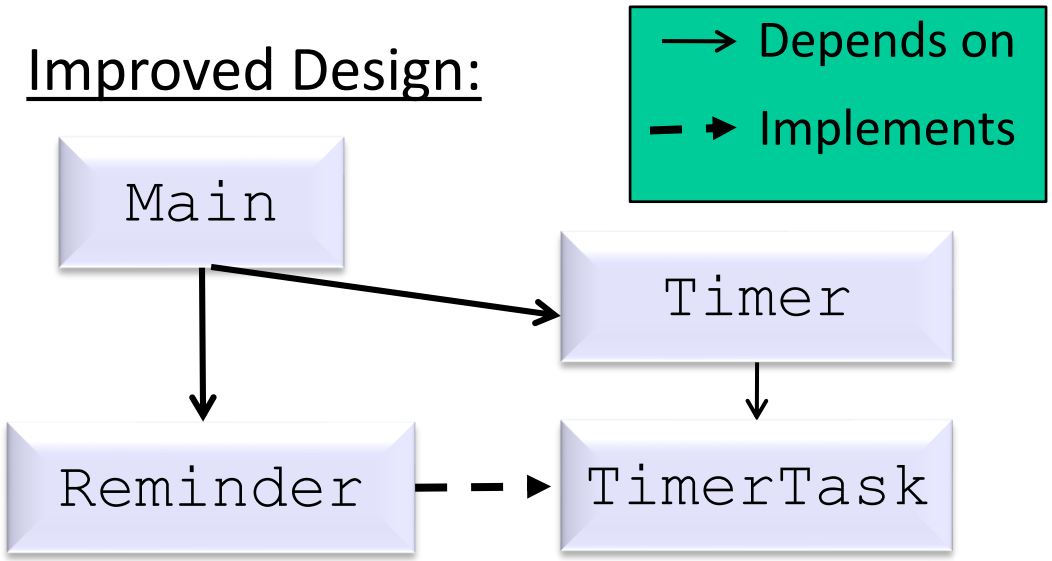
```
…
Reminder reminder = new Reminder();
reminder.setOn();
```

Let's compare ALL designs

Naïve Design:



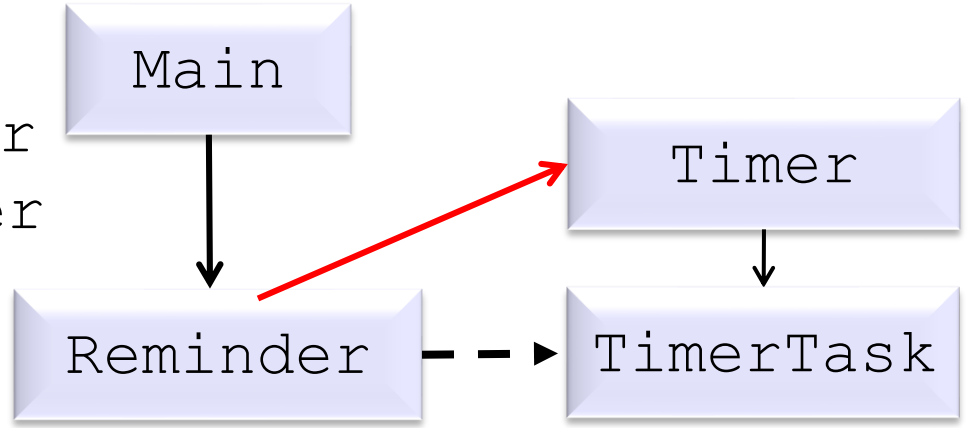
Improved Design:



CALLBACK Design:

`Main` does not depend on `Timer`

`Reminder` depends on `Timer`



Pay attention…

Decoupling and Design:

- While you design (*before* you code), examine dependencies

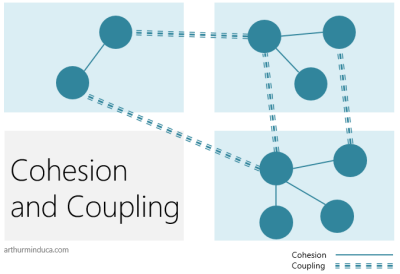
→ Do NOT introduce unnecessary coupling

- Coupling is easy if you code first
  - If a method needs information from another object

  → get it in a modular way
  - Unnecessary coupling (e.g., `Timer` depends on `Reminder`) will:
    - Damage the code's modularity and reusability
    - Yield a more complex code, which is harder to understand

Coupling From Worst to Best

Coupling - the degree to which a class interacts with or depends on other classes



- **Content coupling (worst)** - one class depends on internal data or behavior of another.
- **Common coupling** - sharing common data

  → globals /static final are evil.

- **Control coupling** - knowledge about the implementation of others and passing information to control that logic.
- **Stamp coupling** - sharing more data than needed
- **Data coupling (usually, cannot avoid)** - Passing classes as parameter to method calls (loose coupling)

Content Coupling Example:

one class depends on internal data or behavior of another.


```
public class Line {
   private Point start, end;
   ...
   public Point getStart() { return start; }
   public Point getEnd(){ return end; }
}

public class Arch {
   private Line baseline;
   ...
   void slant(int newY)  {
     Point theEnd = baseline.getEnd();
     theEnd.setLocation(theEnd.getX(),newY);
   }
}
```

`Arch` modifies internal data of `Line` (`Point` instances) through `Point` Interface → `Arch` is content coupled to `Line`

Content Coupling Example:

one class depends on internal data or behavior of another.


```
public class Line {
   private Point start, end;
   ...
   public Point getStart() { return start; }
```

Arch is content coupled to Line - it bypasses the interface of `Line` and uses the interface of `Point` instead.

How is that problematic?

If `Line` changes the way it stores data, or adds additional code to handle point updates → the `Arch` class must be updated as well

```
   void slant(int newY)  {
      Point theEnd = baseline.getEnd();
      theEnd.setLocation(theEnd.getX(),newY);
   }
}
```

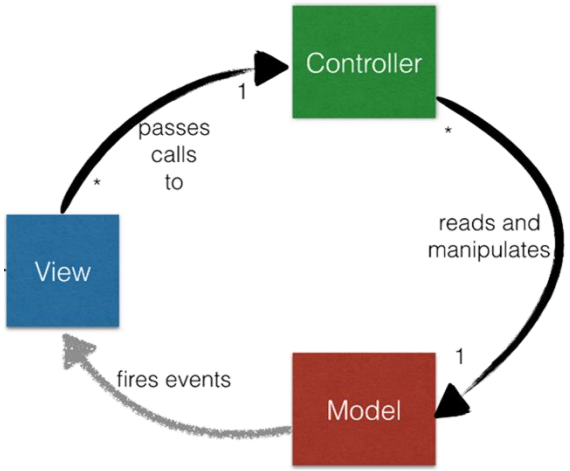
Advantages of reducing connectivity (coupling)

- **Independent development** - decisions made locally, do not interfere with correctness of other modules.
- **Correctness** - proofs easier to derive
- **Potential reusability increased**.
- **Reduction in maintenance costs**
  - Less likely changes will propagate to other modules
  - More robust to errors
- **Comprehensibility** (can understand module independent of environment in which used).

Model-View-Controller Pattern

WHAT?

Division of responsibilities - separating the view layer of your application from the data/logic



WHY?

- Separation of design concerns
- More easily maintainable and extendable
- Promotes division of labor

→ **Model-View-Controller pattern**

- Originated in the Smalltalk community in 1970's
- Widely used in commercial programming
- Recommended practice for graphical applications
- Used throughout Swing (though not obvious on the surface)

MVC Overview

Model

- Contains the “truth” – data/object or state of the system

View

- Visualizes the information in the model to users in desired formats:

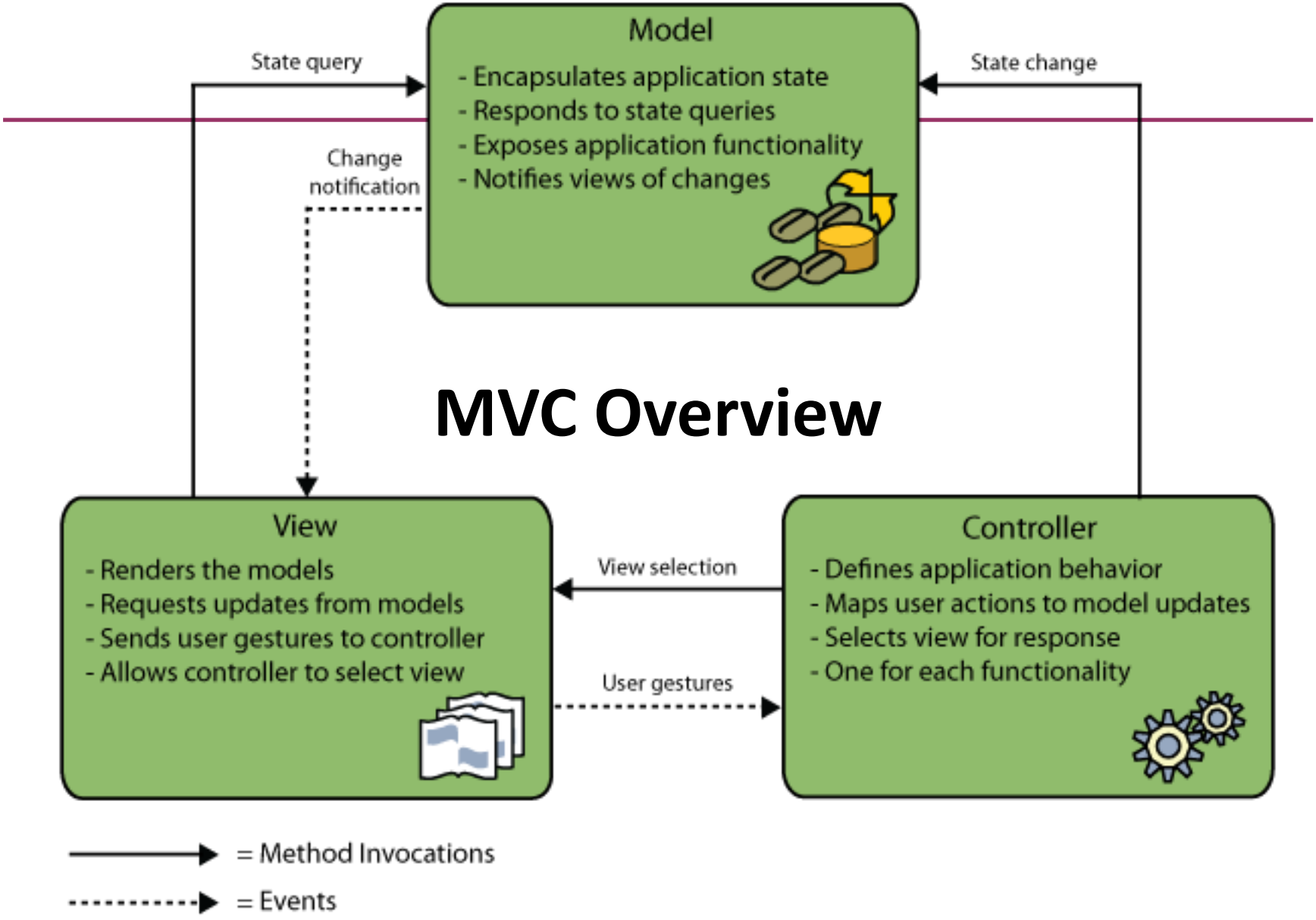
  Graphical display, dancing bar graphs, printed output, network stream….

- Allows for multiple views of a single model

Controller

- Acts on both model and view
- In GUIs, reacts to user input (mouse, keyboard) and other events
- Sends messages to the Model based on events

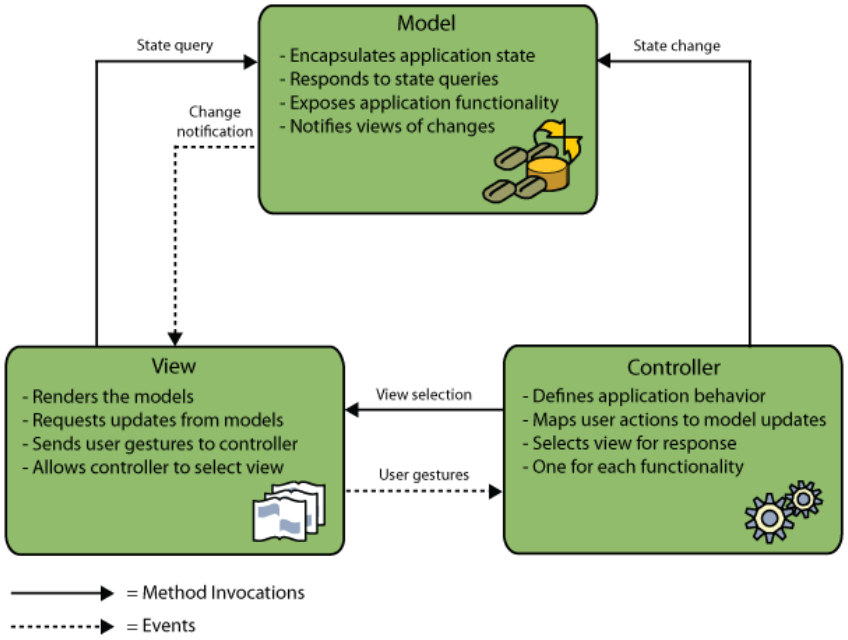
MVC Overview



MVC Interactions and Roles (1)

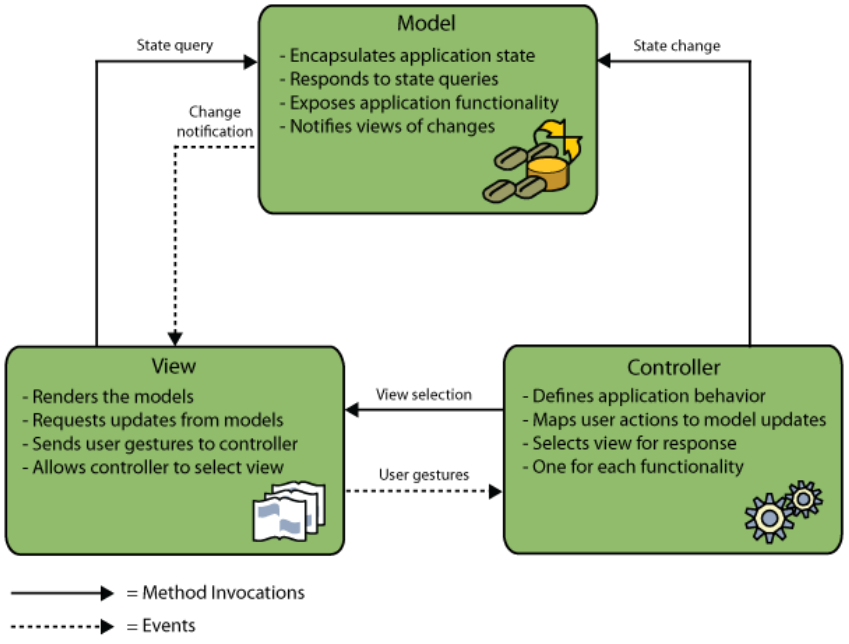
Model

- Encapsulates the data in some internal representation
- Maintains a list of interested viewers
- Notifies viewers when model has changed and view update might be needed
- Supplies data to viewers when requested
- Generally should not know details of the display or user interface details



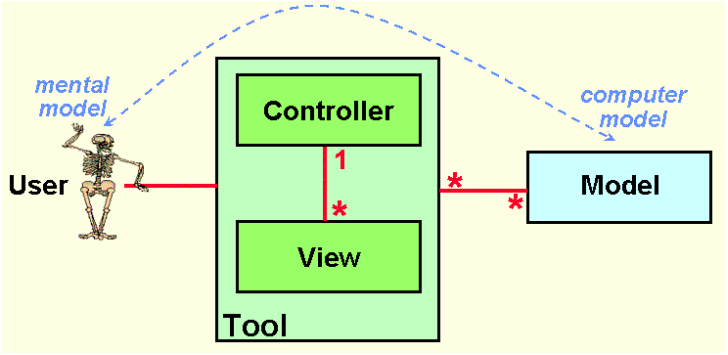
MVC Interactions and Roles (2)

- **View**
  - Maintains details about the display environment
  - Gets data from the model when it needs to
  - Renders data when requested (by the model or the controller)
  - May catch user interface events and notify controller
- **Controller**
  - Intercepts and interprets user interface events
  - Routes information to model and views



MVC vs MV

- Separating Model from View → basic good object-oriented design



- Separating the Controller is a bit less clear-cut
  - Maybe overkill in a small system
  - Often the Controller and the View are naturally closely related

    Both frequently use GUI Components (unlikely for Model)
- Model-View pattern
  - OK to fold the Controller and the View together when it makes sense
  - Fairly common in modern user interface packages

Implementation Note

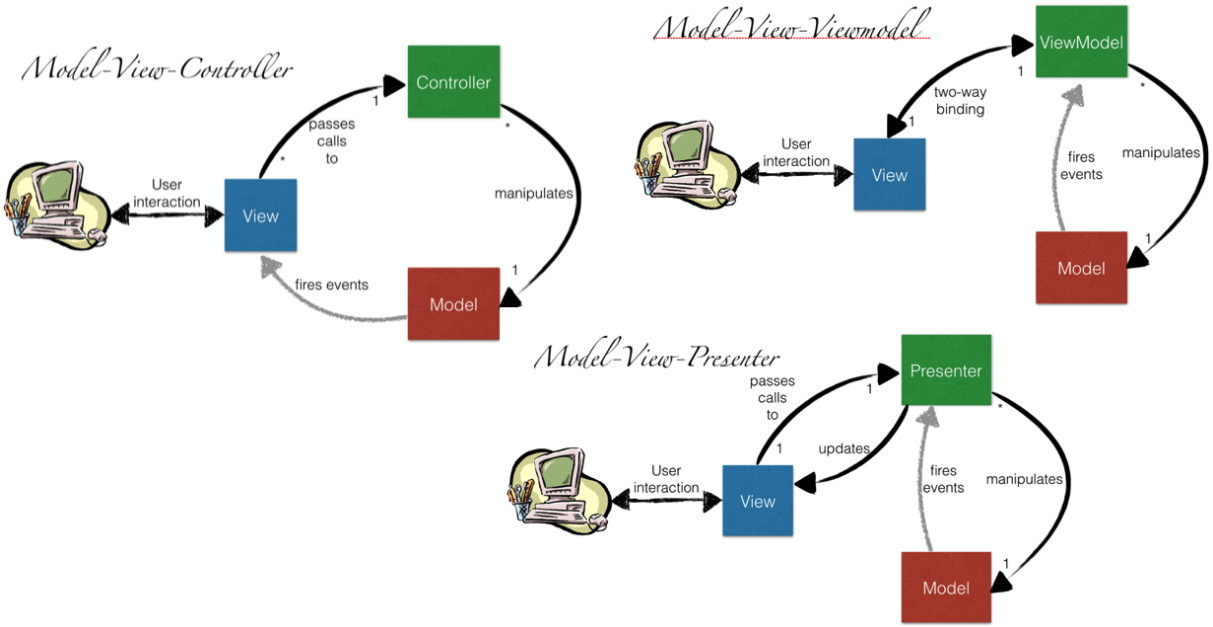
- Model, View, and Controller are **design concepts**, **not** class names

→Might be more than one class involved in each

- Can have multiple views and controllers, BUT only 1 model !
- Models/ views are possibly reusable. Controller is NOT reusable

Moreover…

- What if the view itself has significant state and behavior?
- How do you manage that or test it?
- Ordinary MVC does not have an answer for that, because the view logic is locked away.
- Other patterns - MVP and MVVM are specific adaptations to MVC to address this concern.



The Observer Pattern

- The MVC design is a particular instance of **"Observer" Pattern:**

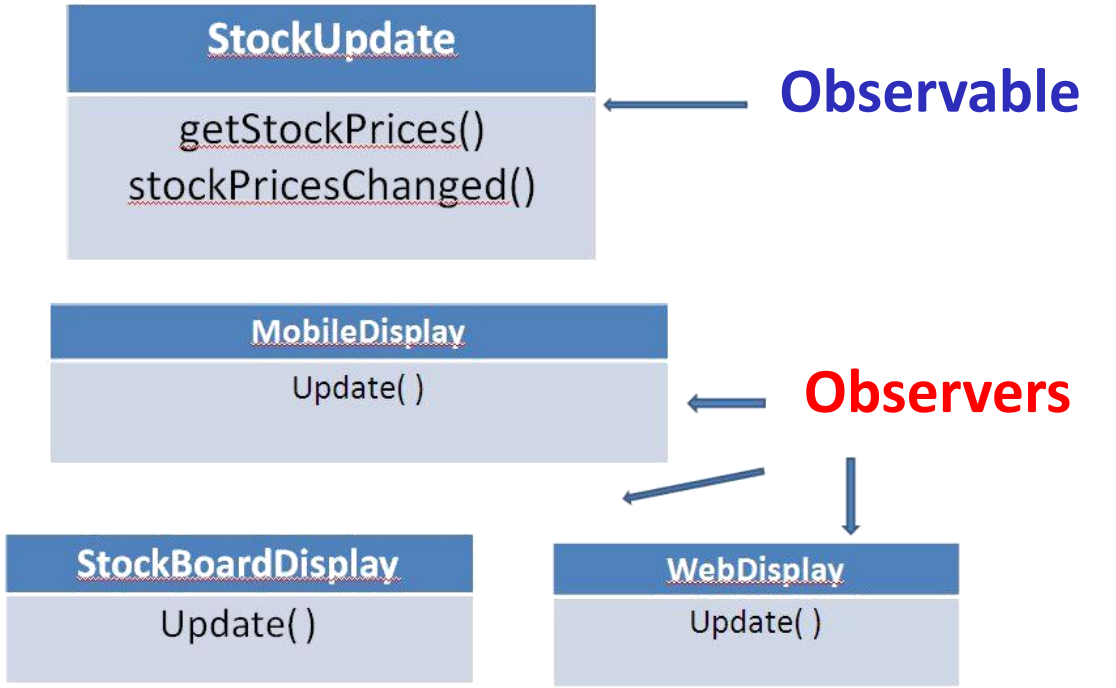
- Object that might change – **observable**, keeps a list of interested **observers** and notifies them when something happens
- Observers can react however they like
- Is used in many areas :
  1. Event and listeners for the `JButton` clicks.
  2. Java RMI (Remote Method Invocation).
- Support in the Java library: interface **`java.util.Observer`** and class **`java.util.Observable`**

If these are a good fit for you → use them
Otherwise → write your own

The Observer Pattern

Observable – monitoring of stock prices (has a state that holds the stock prices)

Observers – waiting for the notifications about the changed stock prices to display the prices on the relevant view



Observer pattern weakens the coupling

What should `StockUpdate` class know about viewers?

→ Observer pattern: call an `update()` method with changed data

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

Any View that is interested to display stock prices must implement this

```
class StockUpdate {
  private PriceInfo priceInfo;
  private List<PriceObserver> observers;
  … // constructor comes here
  void addObserver(PriceObserver newObs) {
    observers.add(newObs);
  }
  void stockPricesChanged(PriceInfo newPriceInfo) {
    this.priceInfo= newPriceInfo;
    notifyObservers();
  }
  void notifyObservers(){
     for (PriceObserver obs : observers)
           obs.update(priceInfo);
  } … }
```

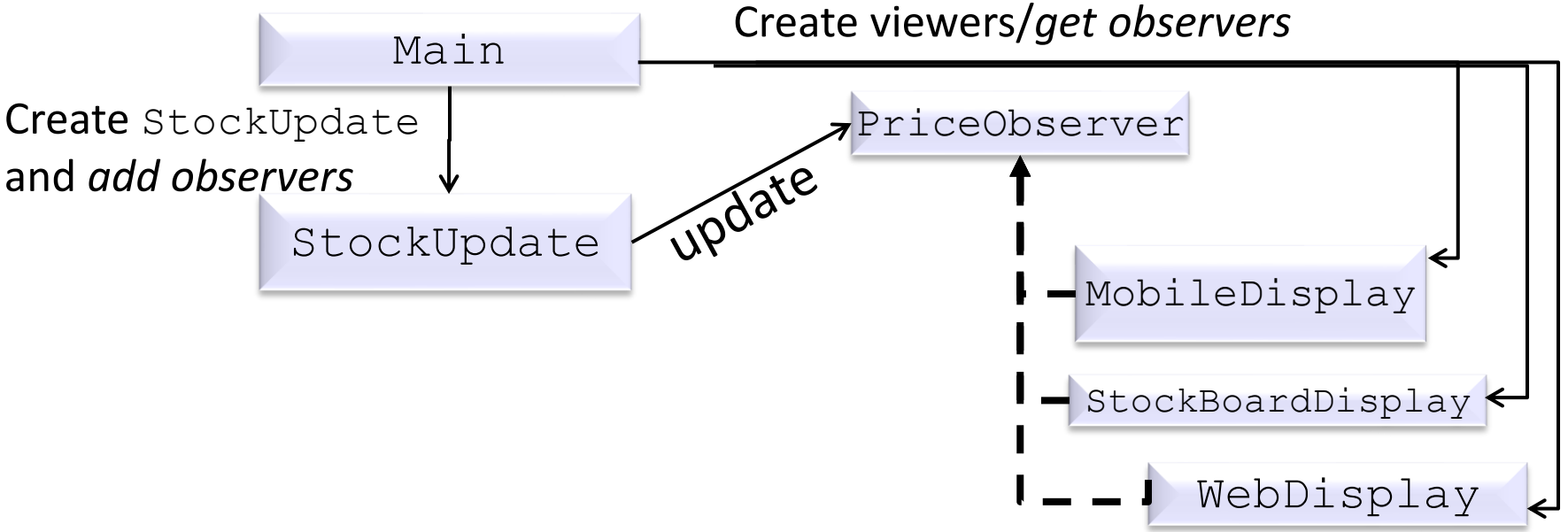
- Encapsulates state
- Maintains list of observers

Register a callback to a new observer

Do the callbacks

The observer pattern

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



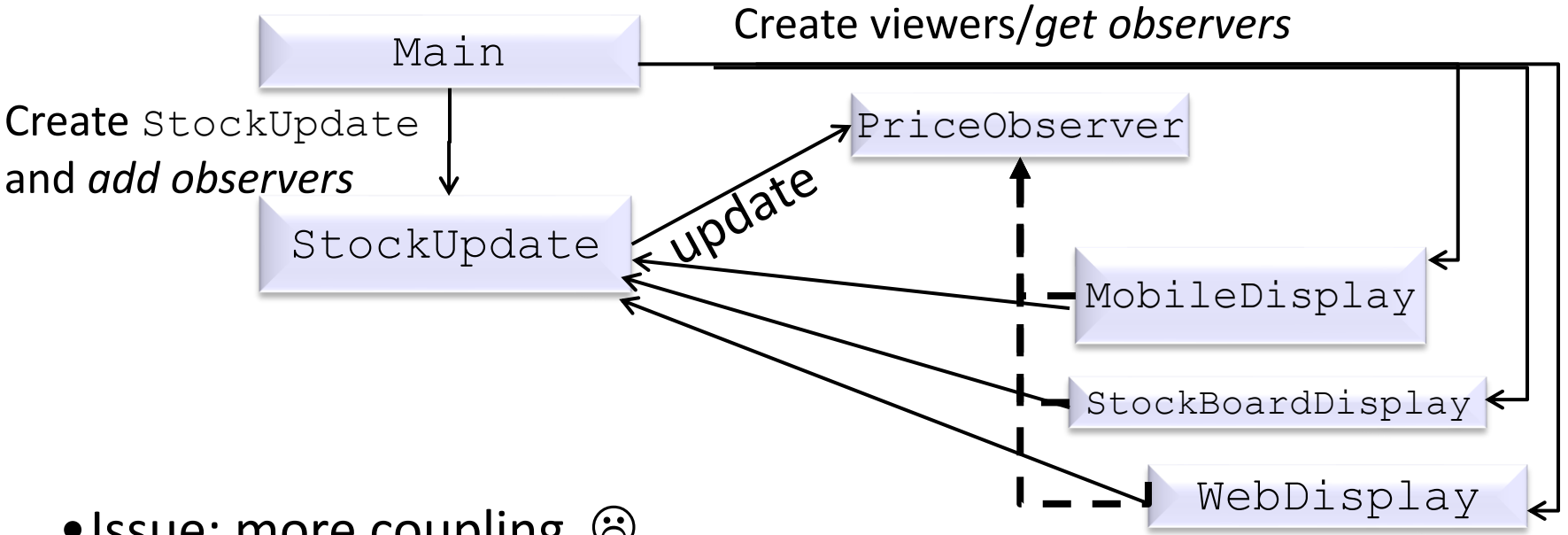
- Issue: `update()` must pass specific information to (unknown) viewers (that otherwise have NO idea about stock prices)

Note on "Push" vs. "Pull"

- Observer pattern implements push models
- Pull model: give all viewers access to `StockUpdate`

→They can extract whatever data they need

→More flexibility ☺



- Issue: more coupling ☹
- Can code both models together

Using JDK Observer

```
// Represents a Course object monitored by ElectronicRoster
public class Course extends Observable {
  private int courseID;
  private String title
  private List<Student> students = new ArrayList<>();
   … //constructor comes here

  public void addStudent(Student student) {
    students.add(student);
    setChanged();
    notifyObservers();
  }
  public int size() {
    return students.size();
  }
  … //getters come here
}
```

Part of the JDK:
Cons: Class WITHOUT interface

Course *inherits* many methods including:

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

Idea: eliminate needless notifications if there is no interesting change to the Observable.
Cons: Forced in all situations

Using JDK Observer

Part of the JDK

```
public class ElectronicRoster implements Observer {
  // called whenever observed object changes
  // and observers are notified
  public void update(Observable o, Object arg) {
    if (o instanceof Course){
        System.out.println("Course" +
            ((Course)o).getTitle() +  "count: "
                + ((Course)o).size());
    }
  }
}
```

Any additional info

Cons: casting ☹ because **Observable** is not generic

Registering an observer

Somewhere in `Main`:

```
Course course= new Course();
course.addStudent(student1);
// nothing visible happens
course.addObserver(new ElectronicRoaster());
course.addStudent(student2);
// now text appears:  "Signup count: 2"
```

At home:

Create you own Observer interface and two observers:

1. View of a new student name that was added to the course
2. View of the total number of students currently enrolled

Note - in this case no need to extend Observable

Practice - anonymous classes and lambda expression (AFTER next class)

Back to Cohesion - God classes

god class: a class that hoards much of the data or functionality of a system

- Poor cohesion – little thought about why all the elements are placed together
- Illusion of coupling reduction - only by collapsing multiple not necessarily related modules into one

A god class is an example of an *anti-pattern*: a known bad way of doing things

Maximizing Cohesion

Methods should do ONE thing well:

- Compute a value but let client decide what to do with it
- Observe or mutate, do NOT do both
- Do NOT print as a side effect of debugging and such

Do NOT limit reusability of the method by having it perform multiple, not-necessarily-related things

"Flag" variables are often a symptom of poor method cohesion
→ Avoid methods that take lots of Boolean "flag" parameters

Method design

Effective Java (EJ) Tip #40: Design method signatures carefully

- Avoid long parameter lists
- "If you have a method with ten parameters, you probably missed some."
- Especially error-prone if parameters are all the same type

Which of these has a bug?

- **memset(ptr, size, 0);**
- **memset(ptr, 0, size);**

EJ Tip #41: Use overloading judiciously

Can be useful, BUT avoid overloading with same number of parameters, and only if methods are really related

Field design

A variable should be made into a field if and only if:

- It is part of the **inherent internal state** of the object
- It has **a value that retains meaning** throughout the object's life
- Its state must persist past the end of any one public method

All other variables can and should be local to the methods in which they are used

- Fields should not be used to avoid parameter passing
- NOT every constructor parameter needs to be a field

There are a few exception to the rule:

- Example: **Thread.run**

Constructor design

Constructors should have all the arguments necessary to initialize the object's state – no more, no less

Object should be completely initialized after constructor is done

Client should NOT need to call other methods to “finish” initialization

Constructors CAN be private/protected

Enums and Abstraction

- Consider use of **enums**, even with only two values – which of the following is better?

```
oven.setTemp(97, true);
oven.setTemp(97, Temperature.CELSIUS);
```

- Consider creating a Type for complex data or data that can change representation. For example:
  - Urgency (Assignment 2), could be rated using numbers/ letters/words → encapsulate this within a class

  `Enum` is actually a class → check it out!

  But it is LESS flexible. For example: `compareTo()` is final (cannot be overridden)

Class design ideals

Beyond cohesion and coupling…

Completeness: Every (major) class should present a complete interface (self-contained)

Consistency: In names, param/returns, ordering, behavior and exception declaration

- If you have variables that capture time → use the same type
- If you declare runtime exception with `throw` in the signature → do it always
- Are not these confusing?

  `String.length()`, `array.length`, `collection.size()`

Completeness

Include *important* methods to make a class easy to use

Counter examples:

- A mutable collection with **`add`** but no **`remove`**
- A tool object with a **`setOn`** method to select it, but no **`setOff`** method to deselect it
- **`Date`** class with no date-arithmetic operations

Also:

- Objects that have a natural ordering should implement **`Comparable`**
- Objects created by you should override **`equals`** (and therefore **`hashCode`**)
- Most objects should override **`toString`** (VERY useful in debugging and IntelliJ)

But…

Do NOT include everything you can possibly think of

- Once included, it stays there forever (even if almost nobody ever uses it)

Do NOT overcomplicate

- You can always add it later if you really need it

"Everything should be made as simple as possible, but not simpler."

- Einstein

Open-Closed Principle

Software entities should be:

- ***Open for extension***

But

- **Closed for Modification**

→ To add NEW features to your system:

- Add new classes or reuse existing ones in new ways
- If possible, do NOT make changes by modifying existing ones. Why?
  - Existing code works and changing it can introduce bugs and errors.

And once more: **Code to interfaces, not to classes**

Example:

accept a **List** parameter, not **ArrayList** or L**inkedList**

EJ Tip #52: Refer to objects by their interfaces

Documenting a class

From EJ:

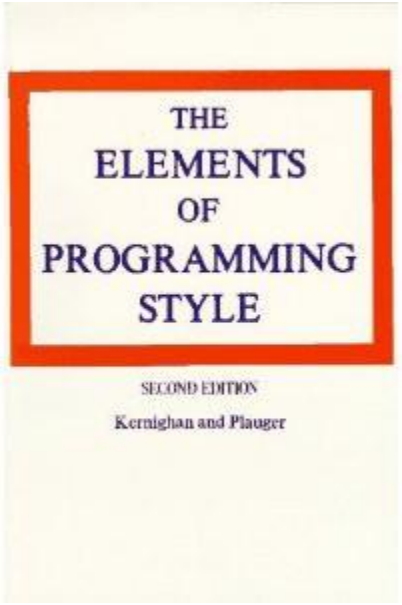
- The doc comment should include all of the method's *preconditions* (things that have to be true before method invocation), and its *postconditions*
- Typically, **preconditions are described implicitly by the `@throws` tags** for unchecked exceptions; each unchecked exception corresponds to a precondition violation.
- Also, **preconditions can be specified along with the affected parameters in their `@param` tags**.
- **Postcondition** checks are best implemented via assertions, whether or not they are specified in public methods.

For example:

```
/*@param index index of element to return; must be non-
 *negative and less than the size of this list
 *@throws IndexOutOfBoundsException if the index is out
 *of range({@code index < 0 || index >= this.size()})*/
```

http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javadoc.html#wheretags
http://ljhs.sandi.net/faculty/Volger/JavaNotes/7.5-PrePostConditions.pdf

The role of documentation
From Kernighan and Plauger



- If a program is incorrect,
  it matters little what the docs say
- If documentation does not agree
  with the code, it is not worth much

→ **Code must largely document itself**.

→ **If not, rewrite the code** rather than increasing the documentation of the existing complex code.

- Good code needs fewer comments than bad code.
- Comments should provide additional information from the code itself. They should not echo the code.
- Meaningful variable names and labels, a layout that shows logical structure, help **make a program "self- documenting"**

Follow MVC Design Patterns

- User interaction should be done through controller/view classes/modules and NOT through classes that maintain the key system data

→ Do NOT put print statements in your core classes

- Instead, return data that can be displayed by the view modules

Which of the following is better?

```
public void printMyself();
public String toString()
```

Use Other Design Patterns

Next week….