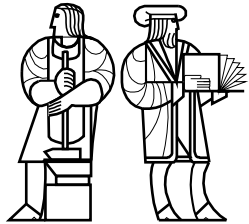


6.170 Lecture 16

Module Dependences and Decoupling



MIT EECS

the human challenge of software

We can imagine assembling hugely intricate structures that work perfectly and indefinitely ...

No friction

No gravity

No wear-and-tear

... but can we **design** them?



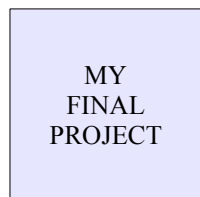
2

the quest for simplicity

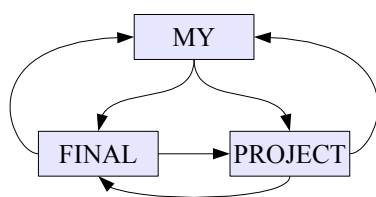
Complexity limits our ability to understand how parts will interact

Key: split design into parts that don't need to interact much

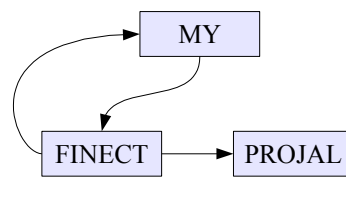
Decoupled parts are simple; **coupled** parts become as complex as the whole



An application



A poor decomposition
(parts strongly coupled)



A better decomposition
(parts weakly coupled)

3

coupling is the path to the dark side

Coupling leads to complexity

Complexity leads to confusion

Confusion leads to suffering

Once you start down the dark path,
forever will it dominate your
destiny, consume you it will



4

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 them 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 multi-threaded solutions for this discussion)

5

TimeToStretch

Assume this is our class to handle making suggestions:

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

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

6

Timer

And here is a class to call run() from time to time:

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

7

Main

Here's how the Main class gets things running:

```
Timer t = new Timer();
t.start();
```

This will work...

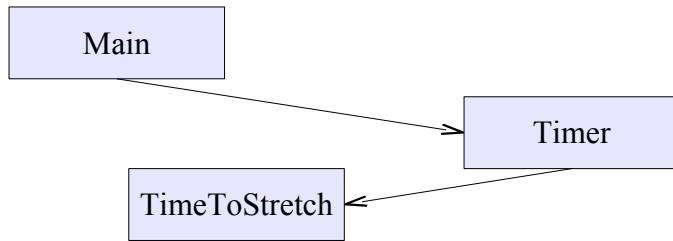
But we can do better

8

module dependency diagram (MDD)

An arrow in a module dependency diagram (MDD) indicates “depends on” or “knows about”

Main class depends on Timer, and Timer in turn depends on TimeToStretch



Does Timer really need to depend on TimeToStretch?

9

decoupling

Timer needs to call the run method

Timer doesn't 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

10

TimeToStretch (version 2)

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

11

Timer (version 2)

Timer is no longer responsible for creating the TimeToStretch object:

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

12

Main (version 2)

Here is how the Main class starts things going:

```
Timer t = new Timer(new TimeToStretch());  
t.start();
```

Now, let's look at our new module dependencies

13

module dependency diagram (version 2)

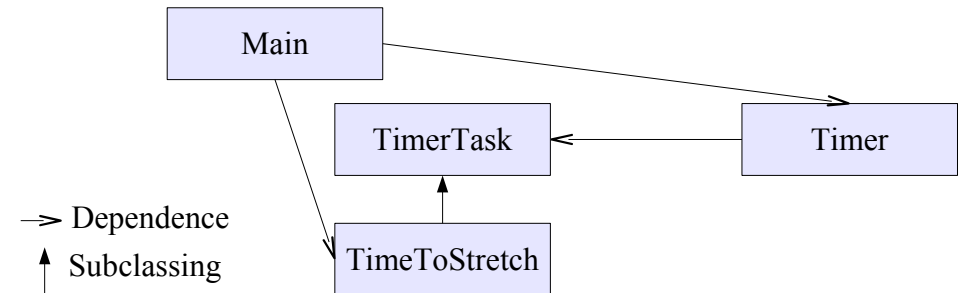
The dependencies have changed

Main depends on Timer

Main depends on the constructor for TimeToStretch

Timer depends on TimeToStretch, but much more weakly

- Unaffected by implementation details of TimeToStretch
- Now Timer is much easier to reuse



14

callback

We can eliminate the dependency of Main on Timer

Make TimeToStretch manage its own Timer

A real version of the application may want to control frequency at which run is called, start/stop timer etc.

Reverse the dependency between Timer and TimeToStretch from our original design

We use a “callback”

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

15

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();  
    }  
    ...  
}
```

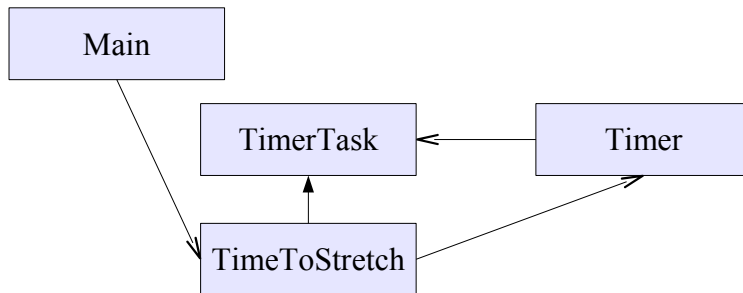
16

Main (version 3)

Here's how we get things running from our Main class:

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

The MDD inverts of the dependency between Timer and TimeToStretch; this is characteristic of callbacks



17

decoupling

An important skill to develop:

- Understand the coupling implied by your design
- Learn to choose that coupling strategically
- We must be able to reason about this *before* writing code
 - We'll try this in the next example

If you rush to code:

- Unnecessary coupling can easily creep in
 - e.g. you realize a method needs some information from some other object, and you hack in a way to get it
 - That hack might be easy to write, but it will generally greatly damage the simplicity and flexibility of your code

18

design exercise #2

Write a program to gather information about stocks from the web, and let the user display this information in many ways

- stock tickers
- graphs
- spreadsheets

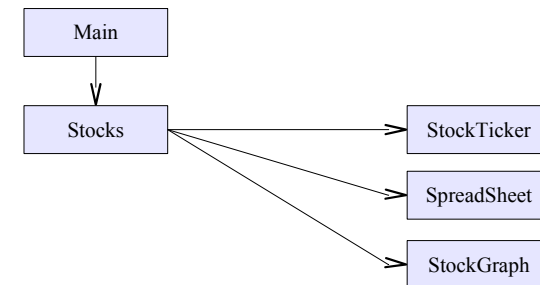
Naive design:

- Make a class to represent stock information
- Make that class responsible for updating all views of that information (tickers, graphs, etc) when it changes

19

module dependency diagram

Main class gathers information and stores in Stocks
Stocks class drives viewers, making sure they get updated when necessary



Drawback: Every time we add or change a viewer, we need to change Stocks. Seems unnecessary – we would like to insulate Stocks from the vagaries of the viewers

20

weaken coupling

Stocks class doesn't need to know much about viewers
Just needs an update method to call when things change

Old:

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

New (uses "observer pattern"):

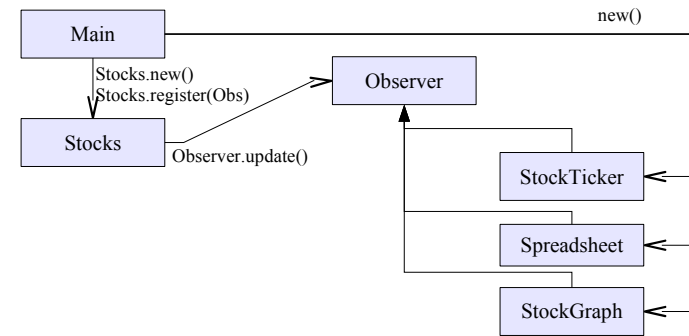
```
List<Observer> observers;  
  
void notifyObserver() {  
    for (Observer obs : observers) {  
        obs.update(newPrice);  
    }  
}  
  
interface Observer {  
    void update(...);  
}
```

How are observers created?

21

the observer pattern

We avoid making Stocks responsible for viewer creation
Main passes them in to Stocks as Observers



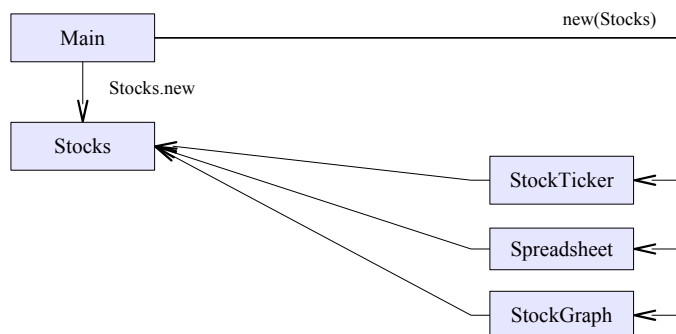
Stocks keeps list of Observers, notifies them of changes
Problem: doesn't know what info each Observer needs

22

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 best design depends on frequency of operations

(It's also possible to use both patterns simultaneously.)

23

concrete example of Observer pattern

Here's a class to store a sign-up sheet of students:

```
public class SignupSheet extends Observable {  
    private List students = new ArrayList();  
    public void addStudent(String student) {  
        students.add(student);  
        notifyObservers();  
    }  
    public int size() {  
        return students.size();  
    }  
}
```

24

concrete example of Observer pattern

Here's an observer for our SignupSheet

See Javadocs for Observer and Observable

```
public class SignupObserver implements Observer {  
    // called whenever the observed object is changed  
    public void update(Observable o, Object arg) {  
        System.out.println("Signup count: " +  
            ((SignupSheet)o).size());  
        // (cast because Observable is non-generic ☹)  
    }  
}
```

25

concrete example of Observer pattern

Here's how our two classes behave:

```
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

26

user interfaces

Adding a user interface may destroy simplicity, flexibility

It is easy to get **appearance** and **content** tangled up

- Particularly when supporting direct manipulation (e.g., dragging line endpoints in a drawing program)

Neither can then be understood easily or changed easily

Over time, leads to bizarre hacks and huge complexity

Huge factor in code longevity

If important program state is stored in widgets in dialog boxes, code will be scrapped very soon

Think carefully!

You'll see callbacks, listeners (and other patterns to come) used everywhere in good code

27

shared constraints

Code dependencies are not the only way for modules to become coupled

A module that writes an image to a file and a module that reads such images from file may not have any dependency on each other's code, but they depend on a common file format

If one fails to write the correct format, the other will fail to read

This is coupling by “shared constraints”

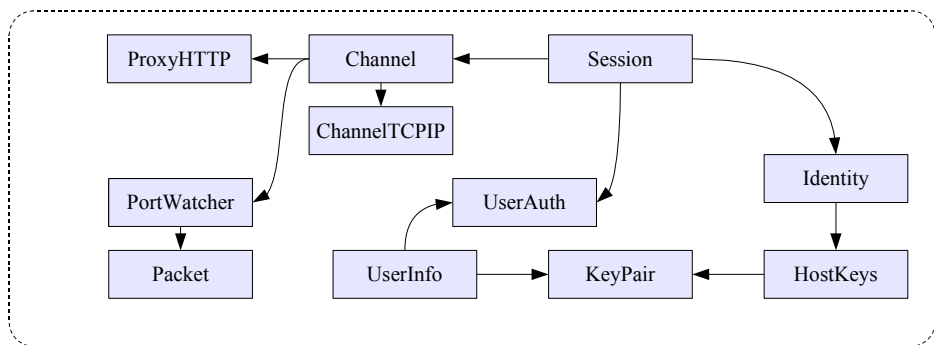
Shared constraints are easier to reason about if they are well encapsulated

Keep the read and the write in a single module that contains and hides all information about the format

28

facade

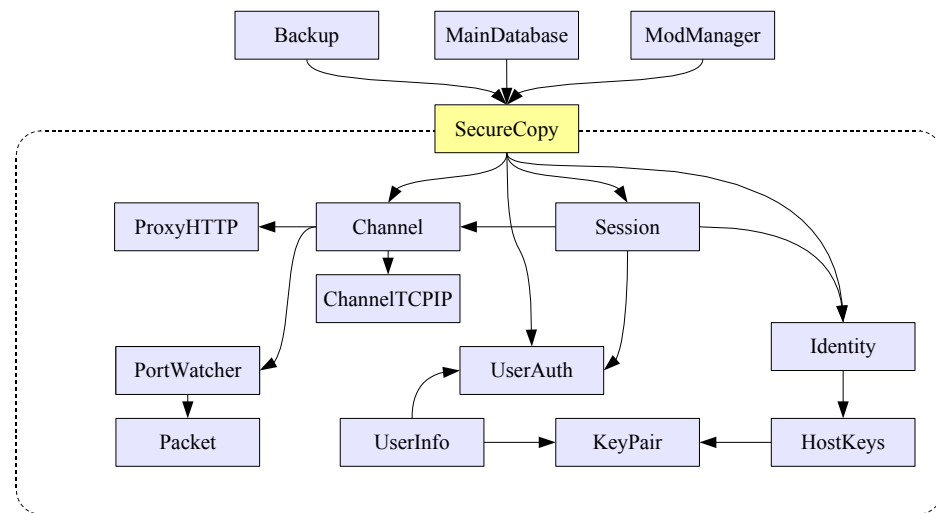
Want to perform secure file copies to a server
Given a general purpose library, powerful and complex
Good idea: build a facade – a new interface to that library that hides its (mostly irrelevant) complexity



29

facade

If library changes, you'll just need to update SecureCopy



30