Module dependences and decoupling

CSE 331

Spring 2010

The limits of scaling

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

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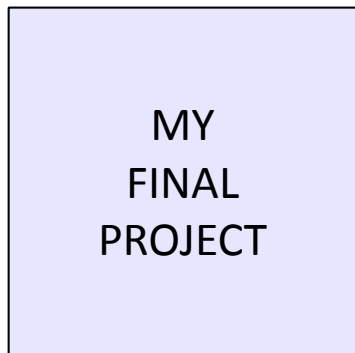- No wear-and-tear

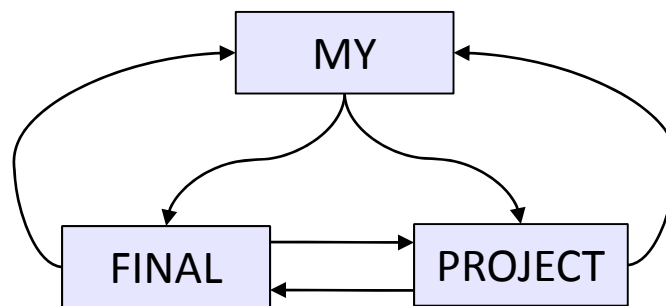… the difficulty of understanding them

Interactions cause complexity

To simplify, split design into parts that don't interact much
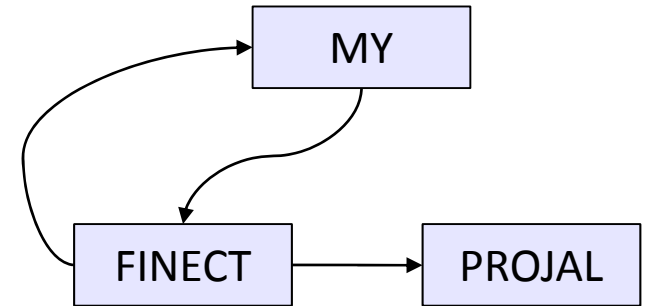
Coupling: amount of interaction between parts

Cohesion: similarity within a part

MY
FINAL
PROJECT

An application

MY

FINAL   PROJECT

A poor decomposition
(parts strongly coupled)

MY

FINECT   PROJAL

A better decomposition
(parts weakly coupled)

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

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 multi-threaded solutions for this discussion)

TimeToStretch suggests exercises

```java
public class TimeToStretch {

    public void run() {

        System.out.println("Stop typing!");

        suggestExercise();

    }

    public void suggestExercise() {

        ...

    }

}
```

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

```java
class Main {
  public static void main(String[] args) {
    Timer t = new Timer();

    t.start();

  }
}
```

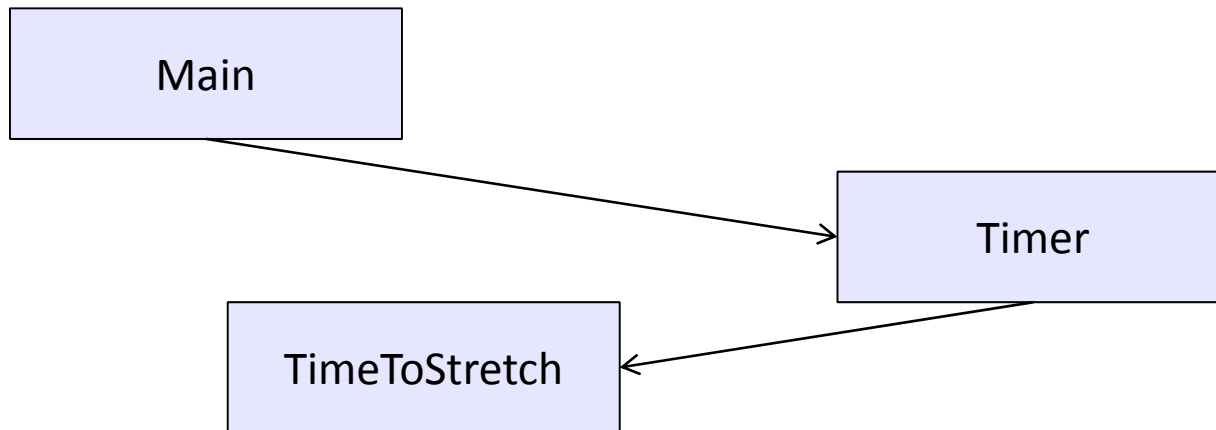This will work…

But we can do better

Module dependency diagram (MDD)

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

Any name mentioned in the source code

Main class depends on Timer

Timer depends on TimeToStretch

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 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

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 the TimeToStretch object:

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

module dependency diagram (version 2)

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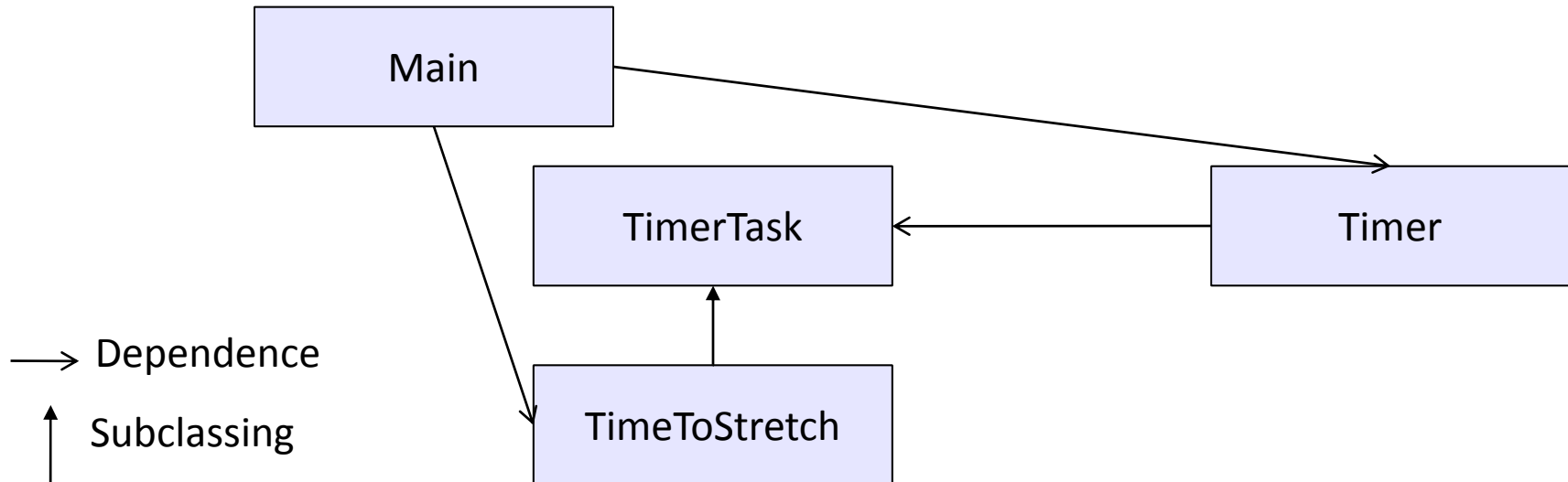
The callback design pattern

- TimeToStretch creates a Timer, and passes in a reference to itself so the Timer can *call it back*
  - A callback is a method call that notifies about some condition
  - To indicate where to call back, *register* interest
- Effects:
  - Main does not depend on Timer
  - TimeToStretch depends on Timer

TimeToStretch (version 3)

```java
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();
    }
    ...
}
```
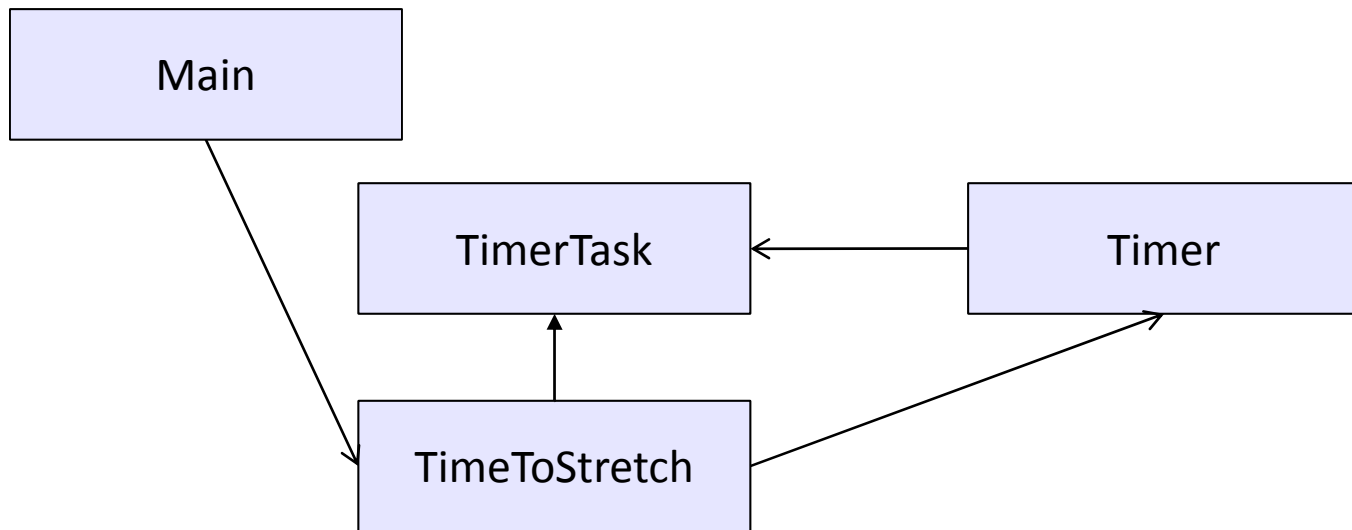
Register interest

Callback entry point

Main (version 3)

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

Use a callback to invert a dependency

This MDD inverts the dependency between Timer and TimeToStretch (compared to ver. 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
  - An easy temptation if you code first
    - 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

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
  - spreadsheets
  - graphs
- 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

Module dependency diagram

Main class gathers information and stores in `Stocks`

`Stocks` class updates viewers when necessary
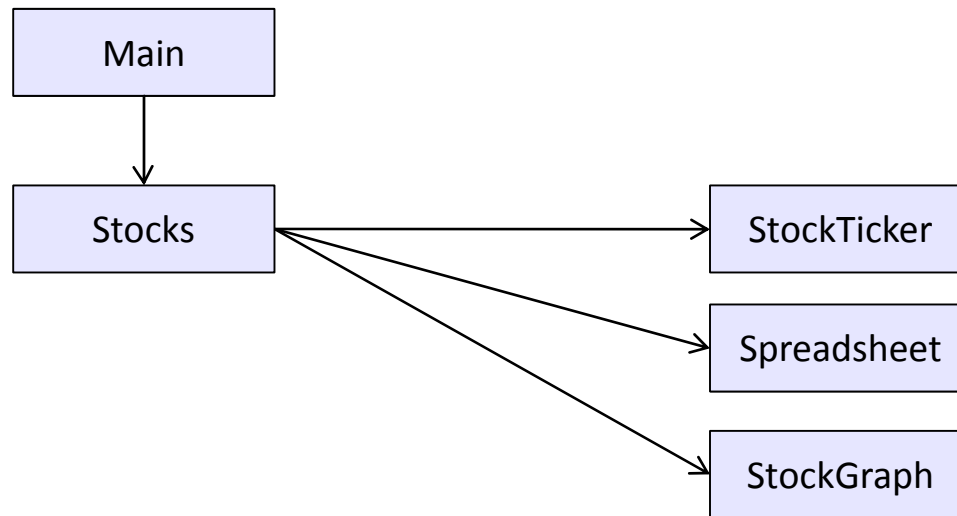
Problem: To add/change a viewer, must change `Stocks`

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

Weaken the coupling

What should Stocks class know about viewers?

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(...);
}
```
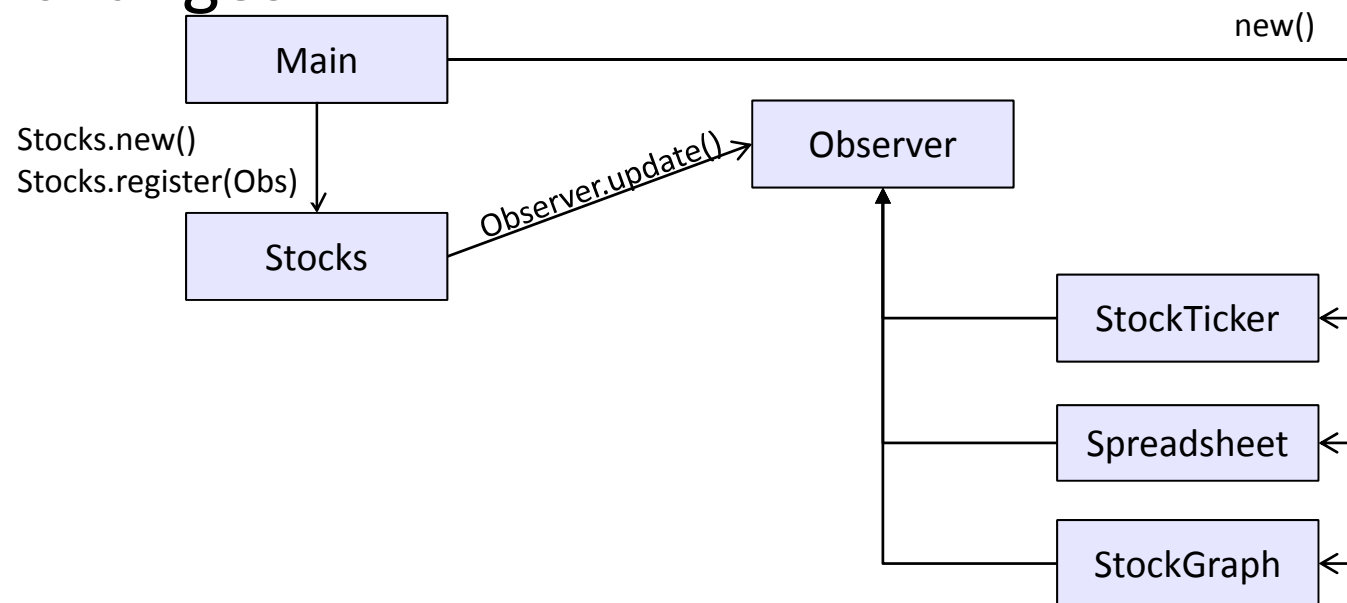
Callback

How are observers created?

The observer pattern

Stocks are not responsible for viewer creation

Main passes viewers to Stocks as Observers

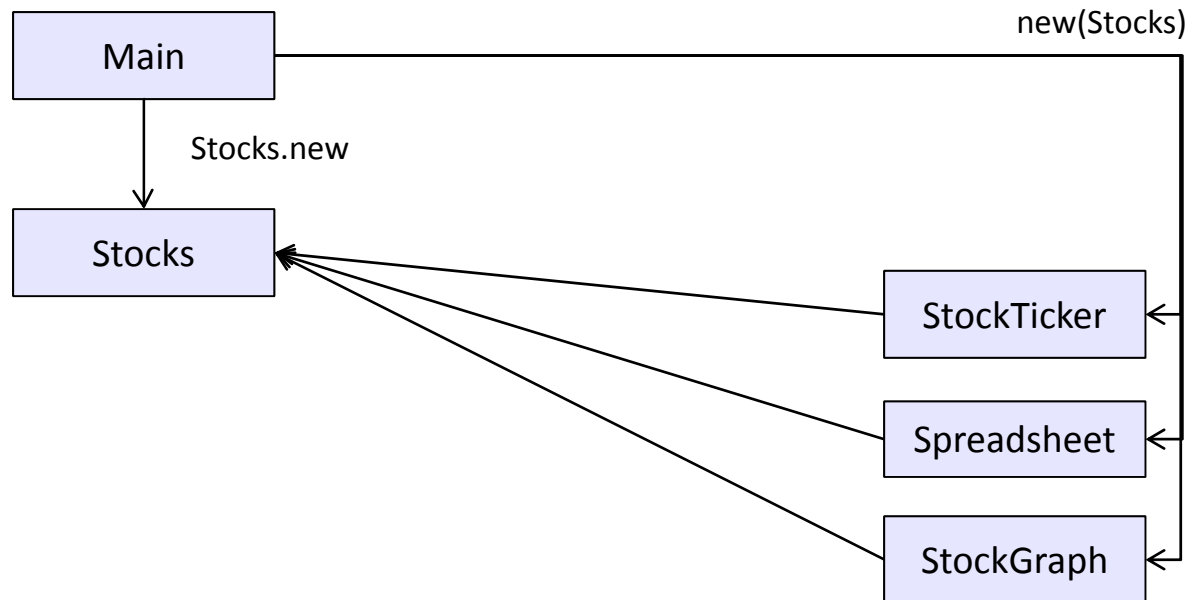Stocks keeps list of Observers, notifies them of changes

Problem: doesn't know what info each Observer needs

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.)

Another example of Observer pattern

// Represents a sign-up sheet of students

```java
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();
  }
}
```

An Observer

```
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 ☹)
  }
}
```

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

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: programstate stored in widgets in dialog boxes

Neither can be understood easily or changed easily

This destroys simplicity and flexibility

- 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
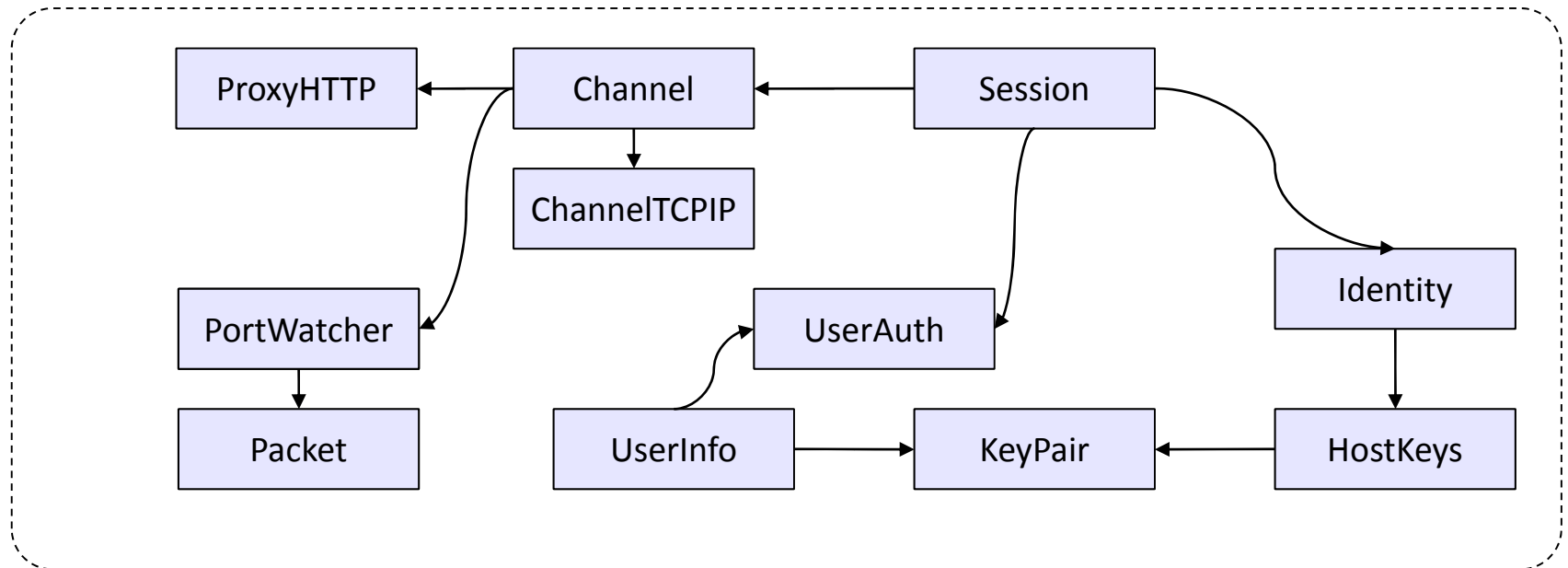
Facade

If library changes, you can update only SecureCopy