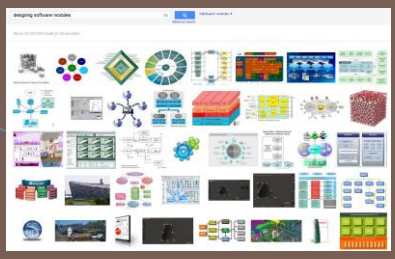
Google image search on "designing software modules"



CSE 331
SOFTWARE DESIGN & IMPLEMENTATION
MODULAR DESIGN PRINCIPLES

Autumn 2011

Dennis Ritchie (1941-2011)

- "'Pretty much everything on the web uses those two things: C and UNIX,' Pike tells Wired. 'The browsers are written in C. The UNIX kernel — that pretty much the entire Internet runs on — is written in C. Web servers are written in C, and if they're not, they're written in Java or C++, which are C derivatives, or Python or Ruby, which are implemented in C.'"
- "'Jobs was the king of the visible, and Ritchie is the king of what is largely invisible,'" says Martin Rinard, professor of electrical engineering and computer science at MIT."
- [Both quotations from CNN]

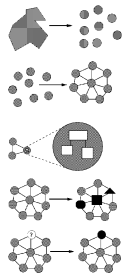
- Above: Ritchie standing, Ken Thompson sitting, PDP-11 in background
- Turing Award. National Medal of Technology. Japan Prize, …

Modules

- A *module* is a 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, but not usually on the implementation of the modules themselves
- Overall, you've been given the modular design so far – and now you have to learn more about how to do the design
  - That's the focus of Assignment #3, and it's why we're using pairs

Ideals of modular software

- Decomposable – can be broken down into modules to reduce complexity and allow teamwork
- Composable – "Having divided to conquer, we must reunite to rule [M. Jackson]."
- Understandable – one module can be examined, reasoned about, developed, etc. in isolation
- Continuity – a small change in the requirements should affect a small number of modules
- Isolation – an error in one module should be as contained as possible



Two general design issues

- *Cohesion* – why are units (like methods) placed in the same module? Usually to collectively form an ADT
- *Coupling* – what is the dependence between modules? Reducing the dependences (which come in many forms) is desirable

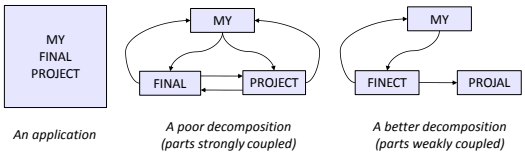
Cohesion

- The most common reason to put elements – data and behavior – together is to form an ADT
  - There are, at least historically, other reasons to place elements together – for example, for performance reasons it was sometimes good to place together all code to be run upon initialization of a program
- The common design objective of separation of concerns suggests a module should address a single set of concerns Example considerations
  - Should Item/DiscountItem know about added discount for purchasing 20+ items? Should ShoppingCart know about bulk pricing?
  - Should BinarySearch know the type of the objects it is sorting?
- This kind of questions help make more effective cohesion decisions

Coupling

Roughly, the more coupled k modules are, the more one needs to think of them as a single, larger module

- How are modules dependent
  - Statically (in the code)? D
    more
  - Ideally, split design into parts that don't interact much



An application | *A poor decomposition (parts strongly coupled)* | *A better decomposition (parts weakly coupled)*

- An artist's rendition – to really assess coupling one needs to know what the arrows are, etc.

Different kinds of dependences

- Aggregation – "is part of" is a field that is a sub-part
  - Ex: A car has an engine
- Composition – "is entirely made of" has the parts live and die with the whole
  - Ex: A book has pages (but perhaps the book cannot exist without the pages, and the pages cannot exist without the book)
- Subtyping – "is-a" is for substitutability
- Invokes – "executes" is for having a computation performed
- In other words, there are lots of different kinds of arrows (dependences) and clarifying them is crucial

Law of Demeter

Karl Lieberherr ⓘ and colleagues

- Law of Demeter: An object should know as little as possible about the internal structure of other objects with which it interacts – a question of coupling
- Or… "only talk to your immediate friends"
- Closely related to representation exposure and (im)mutability
- Bad example – too-tight chain of coupling between classes

```
general.getColonel().getMajor(m).getCaptain(cap)
    .getSergeant(ser).getPrivate(name).digFoxHole();
```

- Better example

```
general.superviseFoxHole(m, cap, ser, name);
```


An object should only send messages to … (More Demeter)

- itself (`this`)
- its instance variables
- its method's parameters
- any object it creates
- any object returned by a call to one of `this`'s methods
- any objects in a collection of the above

Guidelines: not strict rules! But thinking about them will generally help you produce better designs

- notably absent: objects returned by messages sent to other objects

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

God classes

- *god class*: a class that hoards too much of the data or functionality of a system
  - Poor cohesion – little thought about why all of the elements are placed together
  - Only reduces coupling by collapsing multiple modules into one (and thus reducing the dependences between the modules to dependences within a module)

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

Design exercise

- 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

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


Timer calls run() periodically

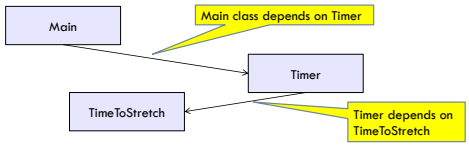
```
public class Timer {
  private TimeToStretch tts = new TimeToStretch();
  public void start() {
    while (true) {
      ...
      if (enoughTimeHasPassed) {
        tts.run();
      }
      ...
    }
  }
}
```


Main class puts it together

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


Module dependency diagram

- An arrow in a module dependency diagram indicates "depends on" or "knows about" – simplistically, "any name mentioned in the source code"



- 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 v2

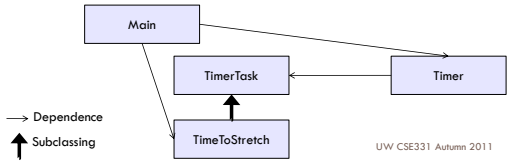
```
public class Timer {
  private TimerTask task;
  public Timer(TimerTask task) { this.task = task; }
  public void start() {
    while (true) {
      ...
      task.run();
    }
  }
}
```

- Main creates the `TimeToStretch` object and passes it to `Timer`

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


Module dependency diagram

- **`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

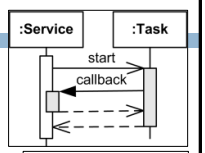


callbacks

- **`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 notifies about some condition
- Use a callback to invert a dependency
  - Inverted dependency: **`TimeToStretch`** depends on **`Timer`** (not vice versa)
  - Side benefit: **`Main`** does not depend on **`Timer`**

Callbacks

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



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

TimeToStretch v3

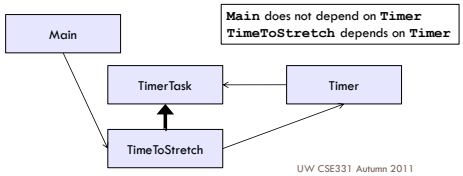
```
public class TimeToStretch extends TimerTask {
    private Timer timer;
    public TimeToStretch() {
        timer = new Timer(this);
    }
    public void start() {
        timer.start();
    }
    public void run() {
        System.out.println("Stop typing!");
        suggestExercise();
    }
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

Register interest with the timer

Callback entry point

Main v3

- `TimeToStretch tts = new TimeToStretch();`
  `tts.start();`
- Use a callback to invert a dependency
- This diagram shows the inversion of the dependency between `Timer` and `TimeToStretch` (compared to v1)



How do we design classes?

- One common approach to class identification is to consider the specifications
- In particular, it is often the case that
  - *nouns* are potential classes, objects, fields
  - *verbs* are potential methods or responsibilities of a class

Design exercise

- Suppose we are writing a birthday-reminder application that tracks a set of people and their birthdays, providing reminders of whose birthdays are on a given day
- What classes are we likely to want to have? Why?

Class shout-out about classes

More detail for those classes

- What fields do they have?
- What constructors do they have?
- What methods do they provide?
- What invariants should we guarantee?

In small groups, ~5 minutes

Next steps

- Assignment 2: part B due today 11:59PM
- Assignment 3: out on the weekend – choose pairs!
  - See http://www.cs.washington.edu/education/courses/cse331/11sp/homework.shtml (HW3, Restaurant) for a preview to get started (2 weeks)
- Assignment 4 and 5: closer to being selected
- Lectures: M – was going to be design patterns… I've had a request for more testing first
- Upcoming: Friday 10/28, in class midterm – open book, open note, closed neighbor, closed electronic devices