

---

CSE 331

# Software Design & Implementation

Kevin Zatloukal

Fall 2020

Module Design & Style

---

# The limits of scaling

---

Can't built arbitrarily large physical structures that work perfectly and indefinitely

- friction, gravity, wear-and-tear

Software has no such problems!

So what prevents arbitrarily large software?

... it's the difficulty of *understanding* it!



The force of friction is replaced by a force in software that creates **interdependence** (“coupling”) between different parts of the code

- in particular, this force makes it hard to understand one part of the code without understanding many other parts

# The limits of scaling

---

Can't built arbitrarily large physical structures that work perfectly and indefinitely

- friction, gravity, wear-and-tear

Software has no such problems!

So what prevents arbitrarily large software?

... it's the difficulty of *understanding* it!

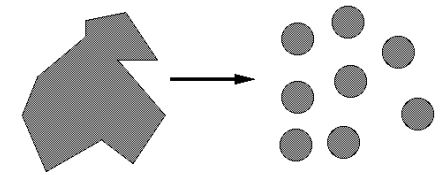
We make software easier to understand by breaking it into pieces that can be understood (and built) separately — using **modularity**



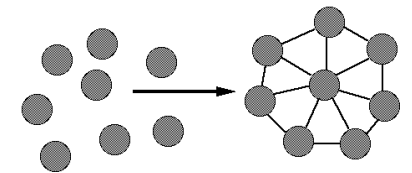
# Many goals of modular software...

---

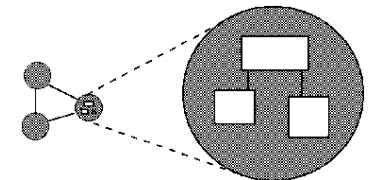
**Decomposable** – can be broken down into modules to reduce complexity and allow teamwork



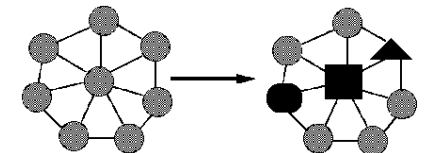
**Composable** – “Having divided to conquer, we must reunite to rule [M. Jackson].”



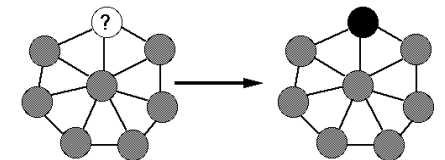
**Understandable** – one module can be examined, reasoned about, & developed in isolation



**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



# Most important design issues

---

*Coupling* – how much dependency there is between components

- want to understand each component without (much) understanding of the others

*Cohesion* – how well parts of a component fit and work together

- form something that is self-contained, independent, and with a single, well-defined purpose

Goals: *decrease* coupling, *increase* cohesion

Applies to modules and smaller units

- each method should do one thing well
- each module should provide a single abstraction

# Cohesion

---

The common design objective, *separation of concerns*, suggests a module should represent a single concept

- a common kind of “concept” is an ADT

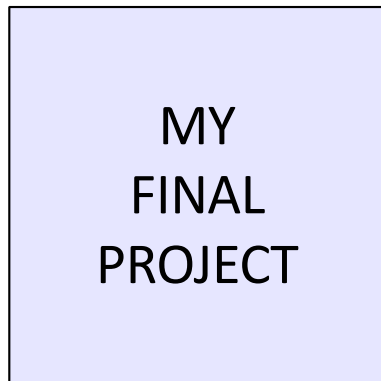
If a module implements more than one abstraction, consider breaking it into separate modules for each one

# Coupling

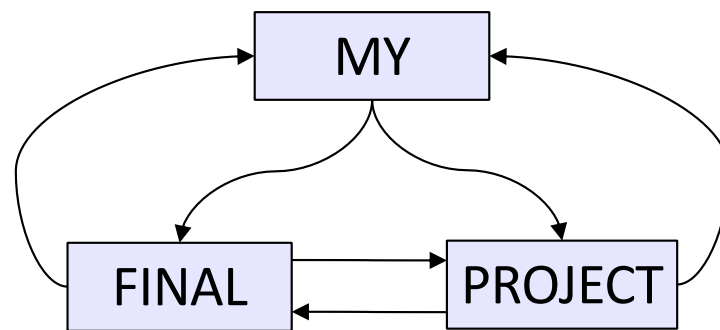
---

How are modules dependent on one another?

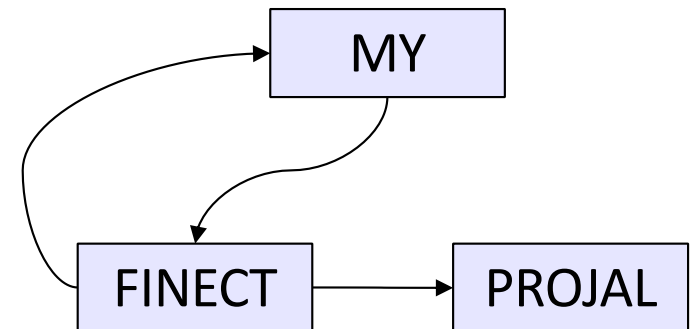
- statically (in the code)? dynamically (at run-time)? for us:
  - do we need to **understand** one to understand the other?
- ideally, split design into parts with little interdependency



*An application*



*A poor decomposition  
(parts strongly coupled)*



*A better decomposition  
(parts weakly coupled)*

The more coupled modules are, the more they need to be thought about all at the **same time** in order to be understood

# Coupling leads to Spaghetti Code

---

- Coupling induces more and more coupling eventually turning into "spaghetti code"
- Lacks all the properties of high quality code
  - hard to understand
  - hard to change
  - hard to make correct
- Often necessary to **throw away** the code and start over





# “God” classes

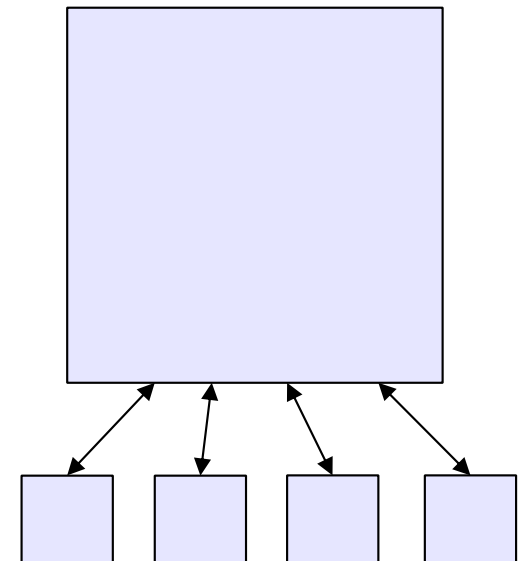
---

*god class*: hoards most of the data or functionality of a system

- depends on and is depended on by every other module
- poor cohesion – little thought about why all the elements are placed together
- reduces coupling but only by collapsing multiple modules into one (which replaces dependences between modules with dependences within a module)

A god class is an example of an *anti-pattern*

- a known **bad** way of doing things



# DESIGN IN JAVA

# Class design ideals

---

Cohesion and coupling, already discussed

*Completeness*: should every class present a complete interface?

- good advice for **public libraries**
- for other code, better to ***avoid unnecessary work***
  - can leave TODOs for what you want to add later
  - or have methods that throw `RuntimeException("not yet implemented")`

*Consistency*: in names, param/returns, ordering, and behavior

- (more later...)

# But...

---

- Don't* include everything you can possibly think of
- if you include it, you're stuck with it forever (even if almost nobody ever uses it)

Tricky balancing act: include what's useful, but don't make things overly complicated

- you can always add it later if you really need it
- except for public libraries, better to wait if you can
  - less code is thrown away when you realize it's all wrong

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

- Einstein

# Example: separate UI from rest

---

- Confine user interaction to a core set of “view” classes and isolate these from the classes that maintain the key system data
  - see Model-View-Controller (and HW9)
- Do not put print statements in your core classes
  - this locks your code into a text representation
  - makes it less useful if the client wants a GUI, a web app, etc.
- Instead, have your core classes return data that can be displayed by the view classes
  - which of the following is better?

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

# Documenting a class

---

Keep internal and external documentation separate

External: `/** ... */` Javadoc for classes, interfaces, methods

- describes things that **clients** need to know about the class
- should be specific enough to exclude unacceptable implementations, but general enough to allow for all correct implementations
- includes all pre/postconditions, etc.

Internal: `//` comments inside method bodies

- describes details of how the code is implemented
- information for fellow developer working on this class
  - tricky parts of the code
  - loop and representation invariants
  - important decisions you made

# Cohesion again...

---

Methods should do one thing well:

- compute a value but let client decide what to do with it
- don't print as a side effect of some other operation
- observe or mutate, don't do both

Having a method do multiple, not-necessarily-related things  
*limits future possible uses*

“Flag” variables are often a symptom of poor method cohesion

- often mean the method is doing multiple things

# Method design

---

Effective Java (EJ) Tip: Design method signatures carefully

- avoid long parameter lists
- especially error-prone if parameters are all the same type
- avoid methods that take lots of Boolean “flag” parameters

Which of these has a bug?

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

EJ Tip: Use overloading judiciously

Can be useful, but avoid overloading with same number of parameters, and think about whether methods really are related



# Consistency

---

A class or interface should have consistent names, parameters/returns, ordering, and behavior

Use similar naming; accept parameters in the same order

Counterexamples:

```
setFirst(int index, String value)
setLast(String value, int index)
```

`Date/GregorianCalendar` use 0-based months

```
String methods: equalsIgnoreCase,
                 compareToIgnoreCase;
                 but regionMatches(boolean ignoreCase)
```

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

# 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  
(i.e., the rep invariant should hold)

Shouldn't need to call other methods to “finish” initialization

- sometimes tempting but an easy way to cause bugs
- complex initialization can be done using a “builder” pattern
  - (more on this in later in the course)

# Field design

---

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

- 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

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

Exception to the rule: when we don't control the interface

- example: **Thread.run**

# Choosing types – some hints

---

Numbers: favor `int` and `long` for most numeric computations

EJ Tip: avoid `float` / `double` if exact answers are required

Classic example: money (round-off is bad here)

Strings are often used since much data is read as text, but keeping numbers as strings is a bad idea.

# Enums make code more readable

---

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

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

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

# Last thoughts (for now)

---

- Always remember your reader
  - Who are they?
    - Clients of your code
    - Other programmers working with the code
      - (including yourself in 6 weeks/months/years)
  - What do they need to know?
    - How to use it (clients)
    - How it works, but more important, *why* it was done this way (implementers)
- Think about mistakes that might be made (by you or others)
  - if you have enough clients, someone *will* make that mistake
  - design to prevent or at least catch those mistakes
  - pay special attention to bugs that will be hard to detect

**READABILITY**

# Naming

---

- Choosing good names is important for **readability**
- With well chosen names, code can be “self-documenting”
  - no need to include comments with explanation
  - code explains itself



# Bad names

---

`flag`, `status`, `compute`, `check`, `pointer`,  
names starting with `my`...

- convey very little useful information!
- (`count` is okay if meaning is *very* clear from context)

Describe what is being counted, what the “flag” indicates, etc.

`numStudents`, `courseIsFull`, ... (fields)  
`calculatePayroll`, `validateWebForm`, ... (methods)

**But** short names in local contexts are good:

Good: `for (i = 0; i < size; i++) items[i]=0;`

Bad: `for (theLoopCounter = 0;  
          theLoopCounter < theCollectionSize;  
          theLoopCounter++)  
      theCollectionItems [theLoopCounter]=0;`

# Good names

---

EJ Tip: Adhere to generally accepted naming conventions

- Class names: generally nouns
  - start with a capital letter (unlike fields & variables)
  - use CamelCaps not Underscore\_Name
- Interface names often –able/-ible adjectives:  
**Iterable, Comparable, ...**
- Method names: noun or verb phrases
  - verbs+noun for observers: **getX, isX, hasX**
  - verbs for mutators: **move, append**
  - verbs+noun for mutators: **setX**
  - choose affirmative, positive names over negative ones  
**isSafe** not **isUnsafe**  
**isEmpty** not **hasNoElements**

# Method Bodies

---

- Write method bodies to make them **easy to read**
  - make life easier for your code reviewer
  - (make life easier for yourself when you come back later)
- Break code into nicely sized “paragraphs”
  - i.e., consecutive lines of code with no blank lines
- Put a comment at the top of the paragraph
  - (unless the code is just as readable as the comment)
  - use full sentences and correct English

# Method Bodies Example 1

---

This code computes “edit distance” (see CSE 421)  
Even if you know what it does, it’s hard to follow.

```
for (int i = 0; i < m; i++)
    A[i][0] = i;
for (int j = 0; j < n; j++)
    A[0][j] = j;
for (int i = 1; i < m; i++)
    for (int j = 1; j < n; j++)
        A[i][j] = min(A[i-1][j] + 1, A[i][j-1] + 1,
            (s[i-1] == t[j-1]) ? A[i-1][j-1] : infinity);
return A[m][n];
```

# Method Bodies Example 1

---

Break into smaller paragraphs and explain what each does.

```
// Fill in match costs for empty prefixes.
for (int i = 0; i < m; i++)
    A[i][0] = i;
for (int j = 0; j < n; j++)
    A[0][j] = j;

// Find the match costs between every pair of prefixes.
for (int i = 1; i < m; i++)
    for (int j = 1; j < n; j++)
        A[i][j] = min(A[i-1][j] + 1, A[i][j-1] + 1,
            (s[i-1] == t[j-1]) ? A[i-1][j-1] : infinity);

// Return the least cost to match the whole strings.
return A[m][n];
```

# Method Bodies Example 1

---

Break into smaller paragraphs and comment each one.

```
// Fill in match costs for empty prefixes.
for (int i = 0; i < m; i++)
    A[i][0] = i;
for (int j = 0; j < n; j++)
    A[0][j] = j;

// Find the match costs between every pair of prefixes.
for (int i = 1; i < m; i++) {
    for (int j = 1; j < n; j++) {
        // Least cost way to match s[0:i] to t[0:j] is lowest
        // of three options: (1) ...
        A[i][j] = min(A[i-1][j] + 1, A[i][j-1] + 1,
            (s[i-1] == t[j-1]) ? A[i-1][j-1] : infinity);
    }
}
```

# Method Bodies Example 2

---

This comment is unnecessary (even insulting):

```
// close the reader  
reader.close()
```



A comment should add something. This adds a little:

```
// clean up  
reader.close()
```

But really, the code is fine by itself:

```
reader.close()
```

# Method Bodies Example 3

---

Don't necessarily need to comment each loop.

This has one comment that describes two `for` loops.

```
// Create directed edges between each pair of nodes.
for (Node start : nodes) {
    for (Node end : nodes) {
        if (!start.equals(end)) {
            graph.addEdge(start, end);
        }
    }
}
```

This is a case where writing the invariant in detail makes it *harder* to understand. (Generally true for “do X for each Y” loops.)