CSE 331
Software Design & Implementation

Hal Perkins

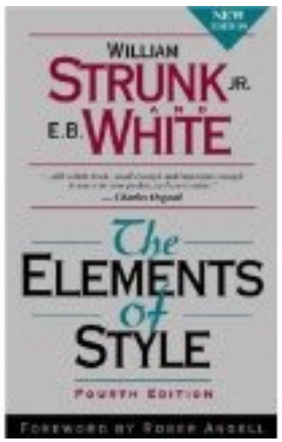
Winter 2023

Module Design and General Style Guidelines

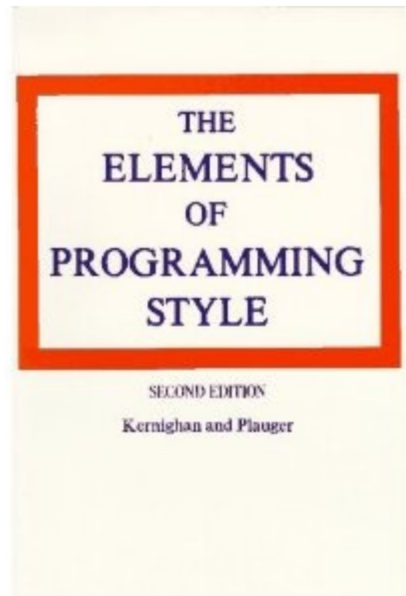
Administrivia

- HW5 part 1 design + tests due Thursday night, 11 pm
  - You should compare ideas with your colleagues and discuss tradeoffs now, then write your own solution
  - Specifications and tests only, no implementations, no rep, no RI/AF, no private methods/classes, …
- Midterm exam next Tuesday night, 2/7, 5-6 pm
  - Rooms TBA
  - Review session Sunday afternoon, details out shortly
  - Topics: everything (lectures, sections, hw assignments, readings) up to and including equals/hashCode
  - Closed book, but you can bring one 5x8 notecard with whatever *handwritten* notes you want on both sides
    - We'll have some blank notecards Wed. to hand out
  - Old exams on web now – might be useful for studying

Style



“Use the active voice.”
“Omit needless words.”



“Don't patch bad code - rewrite it.”
“Make sure your code 'does nothing' gracefully.”

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 problem in software is **interdependence** ("coupling") between different parts of the code

– Coupling makes it hard to understand one part of the code without understanding many other parts and how they all interact

Modules

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

A *module* is a unit in a software system

Class, ADT, package, layer, …

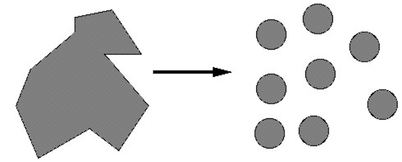
Modular design is the heart of software design

- What modules
- What are their specifications
- How they interact
- But not the implementations of the modules

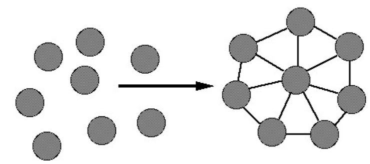
Each module respects other modules' abstraction barriers and enforces its own

Goals of modular design

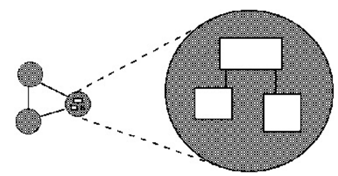
Decomposable – can be broken down into modules to reduce complexity and allow teamwork



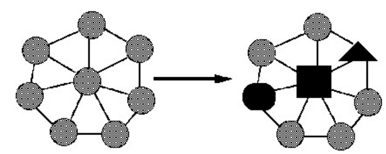
Composable – "Having divided to conquer, we must reunite to rule [M. Jackson]."



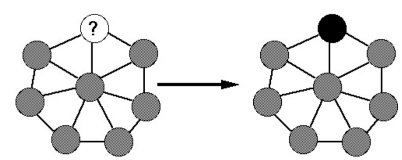
Understandable – one module can be examined, reasoned about, developed, etc. in isolation



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



Most important design issues

Cohesion = internal consistency

- A property of the module specification
  - And applies to implementations
- Want module to be self-contained, independent, and with a single, well-defined purpose

Coupling = dependency between components

- A property of module implementation
- Is usually low when each subpart has good cohesion

Goal: *increase* cohesion, *decrease* coupling

Cohesion

Separation of concerns

For methods: do one thing well

- Compute a value, let client decide what to do with it
- Observe or mutate; don't do both
- Don't print as a side effect of another operation
- "Flag" variables are often a symptom of poor cohesion

For ADTs: provide a single abstraction, represent a single concept

Poor cohesion limits future possible uses

If your module violates this principle, redesign it

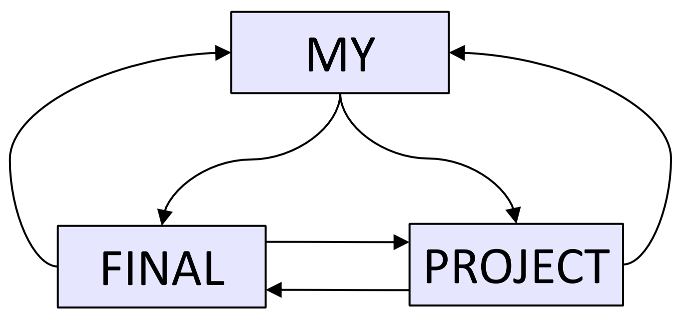
- Refactor a method into multiple simpler methods
- Break an ADT or module into separate ones, each of which represents a single abstraction or concept

Coupling

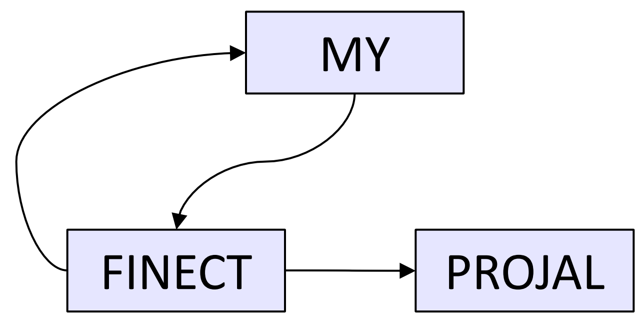
How are modules dependent on one another?

- Statically (in the code)? Dynamically (at run-time)? 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)

If modules are highly coupled you must reason about them as though they are a single, larger module

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 much of the data or functionality of a system

- 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

Class design ideals

Cohesion: already discussed

Coupling: already discussed

Completeness: Every class should present a complete interface

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

Completeness

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

- Especially in public library classes/APIs

  Counterexamples:

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

Also:

- Objects that have a natural ordering should implement **`Comparable`**
- Usually implement (override) **`equals`** (and therefore **`hashCode`**) – more about these in next lecture(s)
- Always override **`Object.toString`** (a superclass may have done this for you)

Don’t include the kitchen sink

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)
- Don’t include compound operations
  (client can call two operations)
- Sometimes use cases mean rethinking completeness: does **`remove`** always make sense for a mutable collection if it is ghastly expensive and never used?

Tricky balancing act that depends on taste

Err on the side of omitting an operation

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

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

Consistency

A module should have consistent names, parameters in the same order, and consistent behavior

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)`**

Collection size:
`String.length()`, **`array.length`**, **`collection.size()`**

Good names

Choosing good names is important (and takes work)

EJ Tip #68: Adhere to generally accepted naming conventions

- Class names: generally nouns describing the type (often ADT) or concept represented by the class
  - Beware "verb + er" names, e.g. **`Manager`**, **`Scheduler`**, **`ShapeDisplayer`**
- Interface names often –able/-ible adjectives:
  `Iterable`, **`Comparable`**, …
- Method names: noun or verb phrases
  - Nouns for observers: **`size`**, **`totalSales`**
  - 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`**

Bad names

`count, flag, status, compute, check, value, pointer`, names starting with **`my`**…

- Convey no useful information

Describe what is being counted, what the "flag" indicates, etc.

`numberOfStudents`, **`courseFull`**, **`noMorePizza`**, **`calculatePayroll`**, **`validateWebForm`**, …

But short names in local contexts are good:

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

Not:
```
for(theLoopCounter =  0;
    theLoopCounter < theCollectionSize;
    theLoopCounter++)
  theCollectionItems[theLoopCounter]=0;
```

Documenting a class

- Keep internal and external documentation *separate*
- External documentation: Specification
  - `/** ... */` Javadoc for classes, interfaces, methods
  - What clients need to know
  - Includes abstract values & invariants, pre/postconditons, etc.
- Internal documentation: Implementation
  - `//` comments inside method bodies & classes
  - Clients don't need this information and shouldn't know (see) it
  - What someone reading the code needs to know to understand it
  - Includes rep. invariant, abstraction function, internal pre/post conditions, algorithm explanations, *rationale* for design and implementation choices, *why* it was done this way
  - If it's hard to document/explain, redesign it

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 between public method invocations

All other variables should be local to a method

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

Exception: Certain cases where overriding is needed

- 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

- The rep invariant should hold

Client 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)

Method design

Effective Java (3rd ed.) Tip # 51: Design method signatures carefully

- Avoid long parameter lists
  - Perlis: "If you have a procedure 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);
```

- Avoid methods that have lots of (or any?) Boolean "flag" parameters

EJ Tip #52: Use overloading judiciously

- Avoids having arbitrary different method names
- But use only when specifications are analogous

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

- Think about whether “paragraphs” should be broken out into separate methods (not always, but sometimes, especially if code is duplicated or reusable elsewhere)

Open-Closed Principle

Software entities should be *open for extension*, but closed for modification

- Add features by adding new classes or reusing existing ones in new ways
- Avoid modifying existing ones
  - Changing existing code can introduce bugs and errors

Related: Code to interfaces, not to classes

Example: accept a **`List`** parameter, not **`ArrayList`** or **`LinkedList`**

EJ Tip #64: Refer to objects by their interfaces

Really: use the most general/highest type that provides the needed operations

Enums improve readability

Consider use of **`enums`**, even with only two values

Which of the following is better?

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

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

(see EJ #51)

Choosing types – some hints

Numbers: Favor **`int`** and **`long`** for most numeric computations

EJ Tip #60: Avoid **`float`** and **`double`** if exact answers are required

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

Avoid using **`String`** representations

- If implementation is parsing String representations, redesign
  - (watch for **`String.indexOf`**, regular expressions)
- **`String`** is tempting because it's a common input/output format, but avoid unless the data actually is text
  - (don't store numbers as strings)
- EJ Tip #12: provide observer methods so client doesn't have to rely on exact format of **`toString()`** output

Independence of views

Confine user interaction to a core set of “view” classes

- Isolate these from the “model” classes that maintain the key system data

Do not put print statements in your core data (model) 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 model classes return data for use by view classes

- Which of the following is better?

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

The model is small

- *Do* keep the core model of what you are doing small and independent

- *Don't* get sloppy on the "extra layers" around it
  - It ends up being most of your code!

Less than 10% of the code has to do with the ostensible purpose of the system; the rest deals with input-output, data validation, data structure maintenance, and other housekeeping.

-- Mary Shaw

Last thoughts (for now)

- Specs and code are read more often than written – writing matters!
- Who are your readers?
  - Clients of your code – need to know how to use it
  - Programmers maintaining the code – need to know how it works, but, even more, *why* it was done this way
    - (including *you* in 3 weeks/months/years)
- Write comments and documentation when you create things – don't try to reconstruct "why" later
- Read/reread style and design advice regularly
- Keep practicing – mastery takes time and experience
- You'll always be learning. Get feedback! Keep looking for better ways to do things!