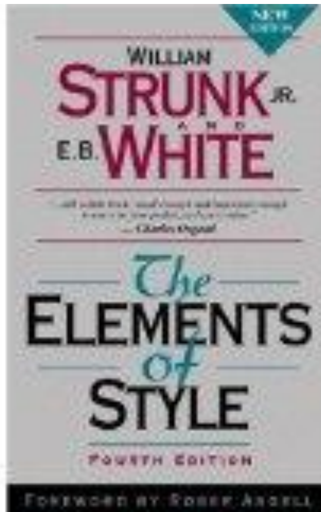
Module design and code style

CSE 331

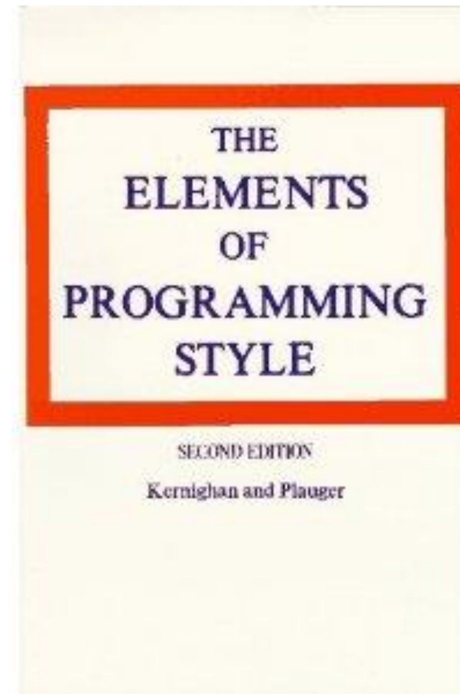
University of Washington

Michael Ernst

Style



"Use the active voice."
"Omit needless words."



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

Modules

- Module: a unit of a software system
  - Class, package, layer
- Designing modules is the heart of software design
  - What modules
  - Their specifications
  - How they interact
  - Implementation is irrelevant to design
- Each module enforces its abstraction barrier

Goals of (modular) design

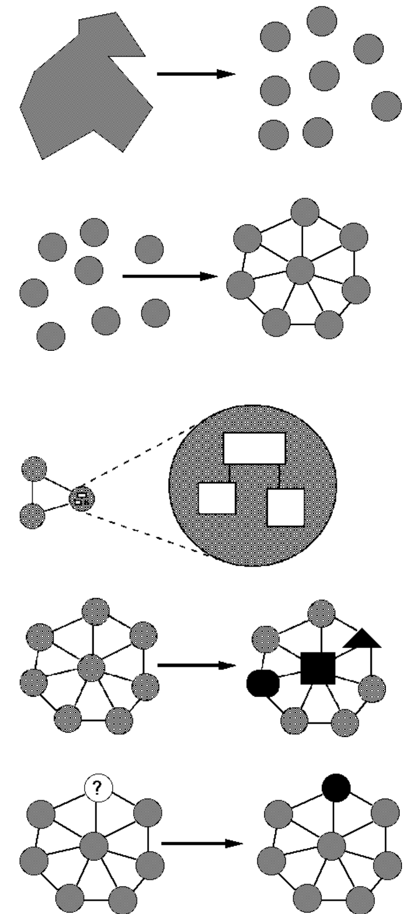
Decomposable – can be broken down into modules to reduce complexity and allow teamwork

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



Separation of concerns: increase cohesion, decrease coupling

Cohesion = internal consistency

A property of a module specification

- Often also applied to an implementation

Is it self-contained, independent, and has a single, well-defined purpose?

Coupling = dependencies between components

A property of a module implementation

Is low when each subpart has good cohesion

Cohesion

Separation of concerns

- For methods: do one thing well
    - Compute a value but let client decide what to do with it
    - Observe or mutate, don't do both
    - Don't print as a side effect of some other operation
    - "Flag" variables are often a symptom of poor method 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

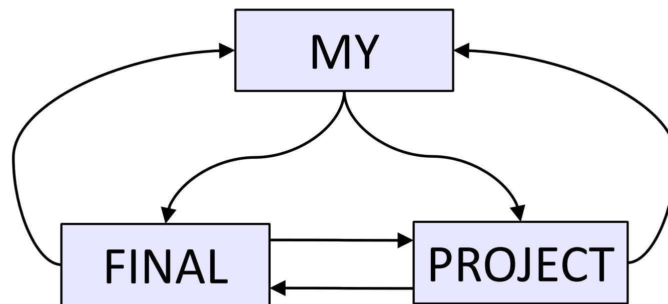
- Refactor a method into multiple simpler methods
- Break an ADT into separate ones, each of which implements one abstraction

Coupling

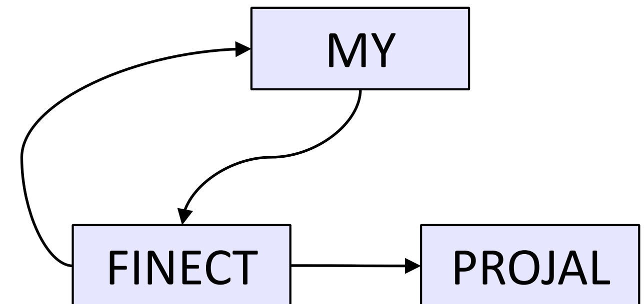
How are modules dependent on one another?

- 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
  - By collapsing multiple modules into one
  - Replaces dependences between modules with dependences within a bigger module

A god class is an anti-pattern: a known bad way of doing things

Method design

Effective Java (EJ) Tip #40: Design method signatures carefully

- Avoid long parameter lists
  - Perlis: "If you have a procedure with ten parameters, you probably missed some."
- Beware of multiple parameters of the same type

  Which of these is correct?

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

- Avoid methods that take lots of Boolean "flag" parameters

EJ Tip #41: Use overloading judiciously

- Useful: don't have arbitrarily-different method names
- Use only if the specification is analogous

Field design

A field:

- Is part of the internal state of the object
- Has a value that retains meaning throughout the object's life
- Its state must persist between public method invocations

Other variables should be local to a method

- Do not use fields 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

- A constructor should fully initialize the object
  - The rep invariant should hold
  - Shouldn't need to call other methods to “finish” initialization
- Constructor should not take any more parameters than necessary to initialize the object's state

Naming

EJ Tip #56: Adhere to generally accepted naming conventions

- Class names: generally nouns
  - 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`**

Names should be informative

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

- Convey no useful information

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

`numberOfStudents`, **`isCourseFull`**, **`calculatePayroll`**, **`validateWebForm`**, …

Use short names in local context:

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

Not:

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

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 or to enable efficient operations
  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
- Objects that have a natural ordering should implement Comparable
- Usually implement equals (and therefore hashCode)
- Always override Object.toString (a superclass might have done this for you)

Don’t include the kitchen sink

- If you include it, you’re stuck with it forever
  - Even if almost nobody ever uses it
- Don’t include compound operations
  - A client can call two operations instead
- A 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 behavior

Violations of this principle:

- **`setFirst(int index, String value)`**
  `setLast(String value, int index)`
- **`Date`** and **`GregorianCalendar`** use 0-based months
- String methods:

  ```
  equalsIgnoreCase
  compareToIgnoreCase
  regionMatches(boolean ignoreCase)
  ```

- Collection size:

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

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 #52: Refer to objects by their interfaces

Really: "Use the most general (highest) type that provides the needed operations"

Documenting a class

Specification (external documentation)

- `/** ... */` Javadoc for classes, interfaces, methods
- What clients need to know
- Includes abstract invariants, pre-/post-conditons

Implementation (internal documentation)

- `//` comments
- Clients don't need this information and shouldn't know it
- Useful for a fellow developer maintaining this class
- Rep invariant, abstraction function, internal pre-/post-conditions, algorithm explanation, *rationale* for design and implementation choices
- "Self-documenting code" is rare
- If it's hard to document, redesign it

Keep the two types of documentation *separate*

Enums improve readability

Consider use of enums, even an enum with only two values

Which of these is more readable?

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

(See EJ #40 [51])

Choose appropriate types

EJ Tip #48: Avoid `float` and `double` if exact answers are required

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

Avoid `String` representations

- If the implementation parses the rep, redesign
  - `String.indexOf`, regular expressions
- `String` is tempting because it's a common input format

Independence of views

- Confine user interaction to a core set of “view” classes
  - Isolate these from the “model” classes that represent data
- Do not put print statements in your 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, model classes return data for use by view classes

  Which of the following is better?

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

Design and code for the reader

- Specs and code are read more often than written
  - By clients: need to know how to use it
  - By maintainers, including future you
    - How it works
    - *Why* it was designed this way (more important!)
- Learn style and design advice, and reread it regularly
- Practice! Mastery requires time and experience
  - Get feedback
  - Learn throughout your career