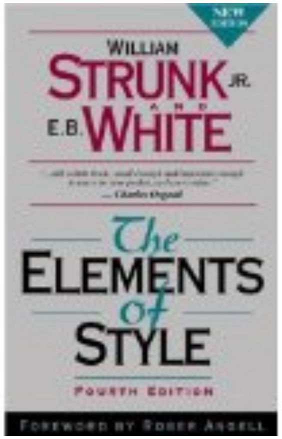
CSE 331
Software Design and Implementation

Lecture 9
Style and Design

Zach Tatlock / Spring 2018

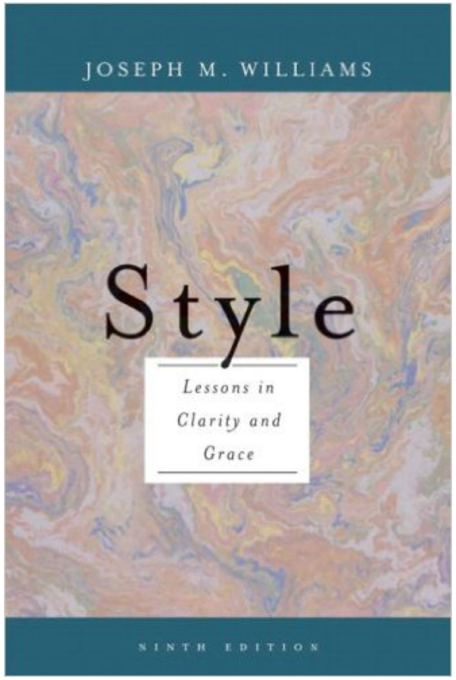
Style: What is a homerun?



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

"Don't patch bad code - rewrite it."

Style: How to hit a homerun



Modules

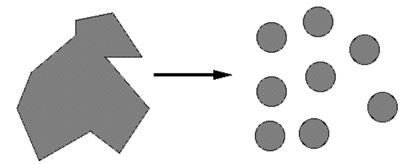
A *module* is a relatively general term for a class or a type or any kind of design unit in software

A *modular design* focuses on what modules are defined, what their specifications are, how they relate to each other

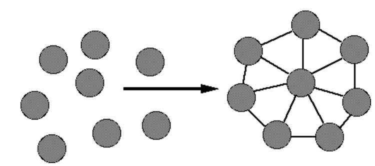
- Not the implementations of the modules
- Each module respects other modules' abstraction barriers!

Ideals of modular software

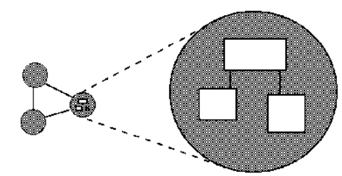
Decomposable – can be broken down into modules to reduce complexity and allow teamwork



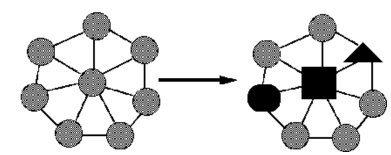
Composable – "Having divided to conquer, we must reunite to rule [M. Jackson]."



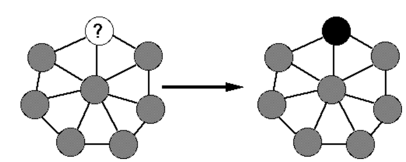
Understandable – one module can be examined, reasoned about, developed, etc. in isolation



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 – how well components fit together to form something that is self-contained, independent, and with a single, well-defined purpose

Coupling – how much dependency there is between components

Guideline: *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 of *separation of concerns* suggests a module should represent a single concept

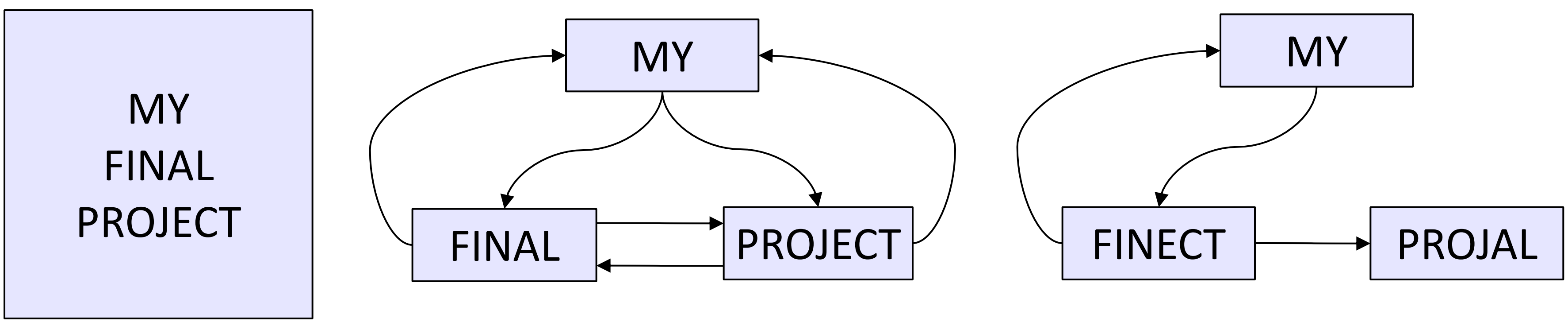
- A common kind of “concept” is an ADT

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)? 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)

Roughly, the more coupled modules are, the more they need to be reasoned about as though they are a single, larger module

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

Cohesion again…

Methods should 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

Don't limit future possible uses of the method by having it do multiple, not-necessarily-related things

"Flag" variables are often a symptom of poor method cohesion

Cohesion vs. coherence

Mary Carillo's Badminton Rant - Athens 2004

Making all the components highly reliable will not necessarily make the system safe.

— **Nancy G. Leveson**

Engineering a Safer World:
Systems Thinking Applied to Safety

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.”
- Especially error-prone if parameters are all the same type
- Avoid methods that take lots of Boolean “flag” parameters

EJ Tip #41: Use overloading judiciously

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

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 past the end of any one public method

All other variables can and should be local to the methods in which they are used

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

Exception to the rule: 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

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

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

Any true wizard knows, once you know the name of a thing you can control it.
-- Jerry Sussman

Good names

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`**

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

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;
```

Class design ideals

Cohesion and 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

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`**
- Objects that might have duplicates should implement **`equals`** (and therefore **`hashCode`**)
- Most objects should implement **`toString`**

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

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

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()`

Open-Closed Principle

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

- When features are added to your system, do so by adding new classes or reusing existing ones in new ways
- If possible, don't make changes by modifying existing ones – existing code works and changing it 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

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/postconditons, etc.

Internal: `//` comments inside method bodies

- Describes details of how the code is implemented
- Information that clients wouldn't and shouldn't need, but a fellow developer working on this class would want – invariants and internal pre/post conditions especially

The role of documentation From Kernighan and Plauger

- If a program is incorrect, it matters little what the docs say
- If documentation does not agree with the code, it is not worth much
- Consequently, code must largely document itself. If not, rewrite the code rather than increasing the documentation of the existing complex code. Good code needs fewer comments than bad code.
- Comments should provide additional information from the code itself. They should not echo the code.
- Mnemonic variable names and labels, and a layout that emphasizes logical structure, help make a program "self-documenting"

Enums help document

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

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

Choosing types – some hints

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

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

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

Strings are often overused since much data is read as text

Independence of views

- Confine user interaction to a core set of “view” classes and isolate these from the classes that maintain the key system data
- 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()
```

Last thoughts (for now)

- Always remember your reader
  - Who are they?
    - Clients of your code
    - Other programmers working with the code
      - (including yourself in 3 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)
- Read/reread style and design advice regularly
- Keep practicing – mastery takes time and experience
- You'll always be learning. Keep looking for better ways to do things!

Large-scale engineered systems are more than just a collection of technological artifacts: They are a reflection of the structure, management, procedures, and culture of the engineering organization that created them. They are usually also a reflection of the society in which they were created.

— **Nancy G. Leveson**

Engineering a Safer World:

Systems Thinking Applied to Safety