CSE 331
Software Design & Implementation

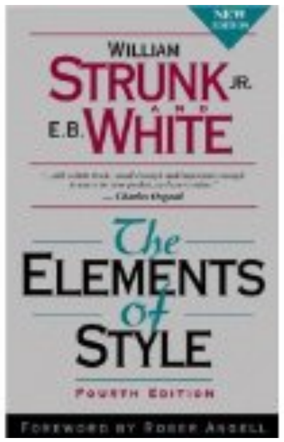
Hal Perkins

Autumn 2013

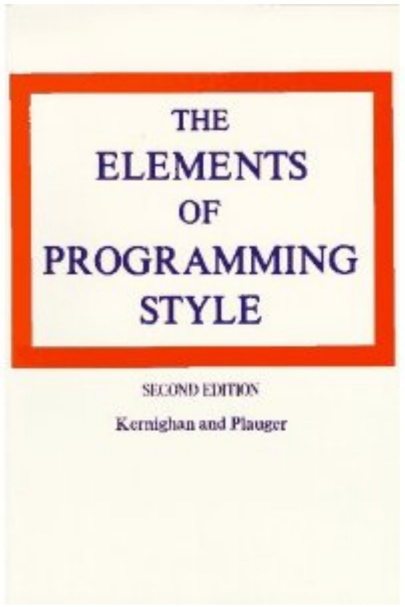
Module Design and General Style Guidelines

(Based on slides by David Notkin and Mike Ernst)

Style: It isn’t just about fashion…



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



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

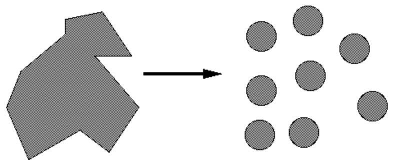
Modules

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

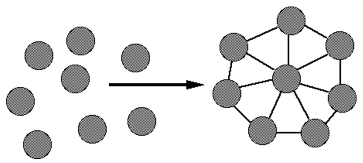
A *modular design* focuses on what modules are defined, what their specifications are, how they relate to each other, but not usually on the implementation of the modules themselves

Ideals of modular software

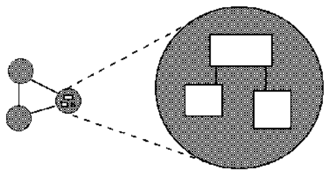
Decomposable – can be broken down into modules to reduce complexity and allow teamwork



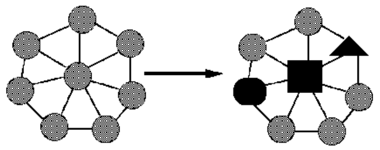
Composable – “Having divided to conquer, we must reunite to rule [M. Jackson].”



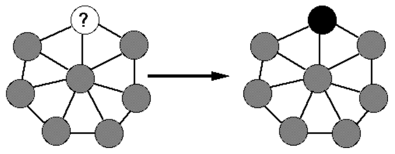
Understandable – one module can be examined, reasoned about, developed, etc. in isolation



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: reduce coupling, increase cohesion

Applies to modules and individual routines

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

Cohesion

The most common reason to put data and behavior together is to form an ADT (data abstraction)

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, but these are secondary

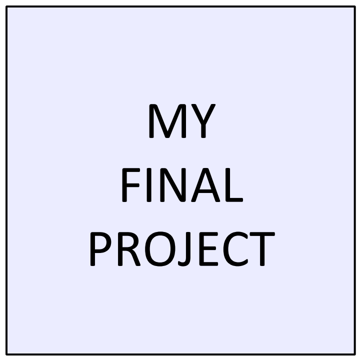
The common design objective of *separation of concerns* suggests a module should represent a single concept

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

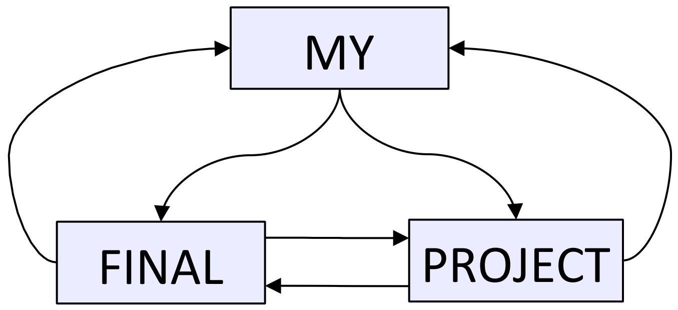
Coupling

How are modules dependent on one another?

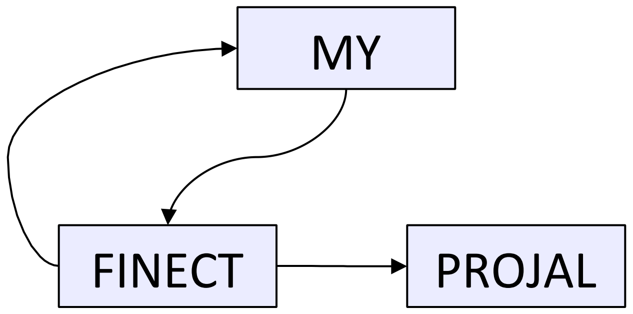
- Statically (in the code)? Dynamically (at run-time)? More?
- Ideally, split design into parts that don't interact much



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 thought of as 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

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 methods' 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

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
- 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* – it is 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 (like printing)

If you've got a method that is doing too much, split it up

- Maybe separate, unrelated methods; maybe one method that does a task and another that calls it
- "Flag" variables are often a symptom of this problem

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

Constructor design

Constructors should take all arguments necessary to initialize the object's state – no more, no less

Don't make the client pass in things they shouldn't have to

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

Minimize the work done in a constructor

- A constructor should not do any heavy work, such as printing state, or performing expensive computations
- If an object's creation is heavyweight, use a **`static`** method instead

Naming

- Choose good names for classes and interfaces
  - Class names should be nouns
    - Watch out for "verb + er" names, e.g. **`Manager`**, **`Scheduler`**, **`ShapeDisplayer`**
    - Interface names often end in -able or -ible, e.g. **`Iterable`**, **`Comparable`**
  - Method names should be noun or verb phrases (nouns for observers, verbs for mutators, etc…)
    - Observer methods can be nouns like **`size`** or **`totalSales`**
    - Many observers should be named with "get" or "is" or "has"
    - Most mutators should be named with "set" or similar
    - Choose affirmative, positive names over negative ones
      - **`isSafe`** not **`isUnsafe`**
      - **`isEmpty`** not **`hasNoElements`**

- EJ Tip #56: Adhere to generally accepted naming conventions

Terrible names…

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

- These convey no useful information
- **`myWidget`** is a cliché – sounds like picked by a 3-year-old
- What others can you think of?

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

- numberOfStudents, courseFull, flightStatus (still not great), 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

Clarity: Interface should make sense without confusion

Convenience: Provide simple ways for clients to do common tasks

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`**)
- Almost all 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 needed/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

Clarity and Convenience

Clarity: An interface should make sense without creating confusion

- Even without fully reading the spec/docs, a client should largely be able to follow his/her natural intuitions about how to use your class – although reading and precision are crucial
- Counterexample: **`Iterator`**'s **`remove`** method

Convenience: Provide simple ways for clients to do common tasks

- If you have a **`size`** / **`indexOf`**, include **`isEmpty`** / **`contains`**, too
- Counterexample: **`System.in`** is terrible; finally fixed with **`Scanner`**

Consistency

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

Use a similar naming scheme; accept parameters in the same order – not like

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

Other examples of inconsistencies:

- **`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 change by modifying existing ones – existing code works and changing it can introduce bugs and errors.

Related: Code to interfaces, not to classes

- Ex: accept a **`List`** parameter, not **`ArrayList`** or **`LinkedList`**
- EJ Tip #52: Refer to objects by their interfaces

Cohesion again ("expert pattern")

The class that contains most of the data needed to perform a task should perform the task

- counterexample: A class with lots of getters but not a lot of methods that actually do work – relies on other classes to "get" the data and process it externally
- countercounterexample: should a graph contain every possible/necessary search algorithm?

Reduce duplication

- Only one class should be responsible for maintaining a set of data, even (especially) if it is used by many other classes

Invariants

Class invariant: An assertion that is true about every object of a class throughout each object’s lifetime

Ex: A **`BankAccount`**'s balance will never be negative

These are often representation invariants

State them in your documentation, and enforce them in your code

Documenting a class

Keep internal and external documentation separate

external: `/** ... */` Javadoc for classes, interfaces, and 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 and abstract class invariants

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

Static vs. non-static design

What members should be **`static`**?

- members that are related to an entire class
- not related to the data inside a particular object of that class's type
- Should I have to construct an object just to call this method?

Examples

- **`Time.fromString`**
- **`Math.pow`**
- **`Calendar.getInstance`**
- **`NumberFormatter.getCurrencyInstance`**
- **`Arrays.toString`**? **`Collections.sort`**?

Public vs. private design

Strive to minimize the public interface of the classes

- Clients like classes that are simple to use and understand
- Reasoning is easier with narrower interfaces and specifications

Achieve a minimal public interface by

- Removing unnecessary methods – consider each one
- Making everything private unless absolutely necessary
- Pulling out unrelated behavior into a separate class

`public static` constants are okay if declared **`final`**

- But still better to have a **`public static`** method to get the value; why?
- Or use enums if that's what you're trying to do

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)

Favor the use of collections (e.g. lists) over arrays

Strings are often overused since much data is read as text

Choosing types – more hints

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

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

Wrapper types should be used minimally (usually with collections)

- EJ Tip #49: Prefer primitive types (**`int, double`**) to boxed primitives (that is, **`Integer`**, **`Float`**, etc.)
  - Bad: **`public Tally(Character ch)`**

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 **`println`** 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!