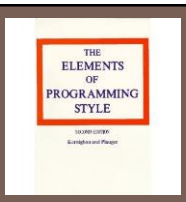
"Use the active voice."
"Omit needless words."



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

CSE 331
SOFTWARE DESIGN & IMPLEMENTATION
STYLE

Autumn 2011

Method design

- A method should do only one thing, and do it well – for example, observe but not mutate, …
- 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 the 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 don't overload with the same number of parameters and think about whether the 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
  - Example: `public Student(String name, int sid)`
    - Why not pass in the student's courses?
- Object should be completely initialized after constructor is done
  - Shouldn't need to call other methods to "finish" initialization
  - NOT: `public Student(String name)`, then calling `setSid(sid)`
- Minimize the work done in a constructor
  - A constructor should not do any heavy work, such as calling `println` to print 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 verb phrases
    - Observer methods can be nouns such as size or totalQuantity
    - 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

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

- Leaving out important methods makes a class cumbersome to use
  - counterexample: A collection with `add` but no `remove`
  - counterexample: A tool object with a `setHighlighted` method to select it, but no `setUnhighlighted` method to deselect it
  - counterexample: `Date` class has no date-arithmetic features
- Related
  - Objects that have a natural ordering should implement `Comparable`
  - Objects that might have duplicates should implement `equals`
  - Almost all objects should implement `toString`

Consistency

- A class or interface should be consistent with respect to 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)`
- Some counterexamples
  - `Date/GregorianCalendar` use 0-based months
  - `String equalsIgnoreCase, compareToIgnoreCase;` but `regionMatches(boolean ignoreCase)`
  - `String.length(), array.length, collection.size()`

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` sucks; finally fixed with `Scanner`

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 – this relies on other classes to "get" the data and process it externally
- 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
- State them in your documentation, and enforce them in your code

Documenting a class

- Keep internal and external documentation separate
- external: `/** ... */` Javadoc for classes 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 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

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 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 you write
  - 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?

Choosing types

- Numbers: Favor **`int`** and **`long`** for most numeric computations
  - EJ Tip #48: Avoid **`float`** and **`double`** if exact answers are required
  - Classic example: Representing money (round-off is bad here)
- Favor the use of collections (e.g. lists) over arrays
- Strings are often overused since much data comes in as text
- 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 to boxed primitives (that is, `Integer`, `Float`, etc.)
    - Bad: `public Counter(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
  - Ex: **`ShoppingMain`**
- 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()`

Next steps

- Assignment 3: out today, pairs assigned, groups created
- Assignment 3: now due Sunday October 30, 11:59PM
- Lectures: W and F (Design Patterns)
- Upcoming: Friday 10/28, in class midterm – open book, open note, closed neighbor, closed electronic devices