CSE 331

Guidelines for Class Design

slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia
http://www.cs.washington.edu/331/

What is class design?

- **class design**: Deciding the contents of a known class (or set of classes) that will effectively solve a given problem.
  - i.e. Classes are told to you (by designer, instructor, etc.) but you have to decide the details of what goes into each class.
  - Differs from *OO design*, which also involves coming up with exactly what classes are needed in the first place.

- Class design references:
  - *Object-Oriented Design Heuristics*, by A. Riel
  - *Object-Oriented Design and Patterns*, by C. Horstmann
  - *Effective Java*, by J. Bloch

Method design

- A method should do only one thing, and do it well.
  - A method should not both access and mutate, except in rare cases.

- **EJ Tip #40**: Design method signatures carefully.
- Avoid long parameter lists (> 4 parameters).
  - If the method needs 7 parameters, maybe something's wrong.
  - Especially prone to errors if the parameters are all the same type.
  - Avoid methods that take lots of `boolean` "flag" parameters.

- **EJ Tip #41**: Use overloading judiciously.
  - **overloading**: Two methods with the same name (different params).
  - 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 always needs to be a field.

  - Sometimes we make exceptions for efficiency (LinkedList size).
  - But do not prematurely optimize. "Caching" values is often bad.

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" initializing it.
  - 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.
    - Accessors methods can be nouns such as size or totalQuantity
    - Most accessors 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 "C" words

Good things that you should strive for when designing classes:

- **1) cohesion**: Every class should represent a *single* abstraction.
- **2) completeness**: Every class should present a complete interface.
- **3) clarity**: Interface should make sense without confusion.
- **4) convenience**: Provide simple ways for clients to do common tasks.
- **5) consistency**: In names, param/returns, ordering, and behavior.

A bad thing that you should try to minimize:

- **6) coupling**: Amount and level of interaction between classes.

1) Completeness

- **completeness**: Every class should present a complete interface.
  - 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`.

Open-Closed Principle

- **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.
    - Reason: Existing code works; changing it can introduce bugs and errors.

- Related: Code to interfaces, not to classes.
  - e.g. accept a `List` parameter, not `ArrayList` or `LinkedList`.
  - **EJ Tip #52**: Refer to objects by their interfaces.

2) Cohesion

- **cohesion**: Every class should represent a *single* abstraction.
  - It should represent *one* thing (not several) and do it well.
  - Keep related data and behavior in one place together.
  - counterexample: `StudentAppointmentScheduler` that keeps track of all info about a student and his/her appointments and schedules them.
  - counterexample: `PokerGame` class that manages all of the players, the chips on the table, the current betting round, computer AI strategies, ...

- Some objects lack cohesion because they are insignificant.
  - Often insignificant objects are better done as `enum`s.
  - Examples: Card suit; Gender; Day of the week

The Expert pattern

- **expert pattern**: The class that contains the majority of the data needed to perform a task should perform the task.
  - counterexample: A class with lots of getters (accessors), not a lot of methods that actually do work.
    - Relies on other classes to "get" the data and process it externally.

- Ostrachan's Law: *"Ask not what you can do with an object; ask what an object can do for itself."*

- Avoid duplication.
  - Only one class should be responsible for maintaining a set of data, even if that data is used by many other classes.

3) Clarity; 4) 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.
  - 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: Java arrays (no behavior)
  - counterexample: `System.in` sucks; finally fixed with `Scanner`
  - counterexample: `Collections` class has to fix flaws in `Lists`

5) Consistency

- **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.
    - bad: setFirst(int index, String value) and setLast(String value, int index) .
  - counterexample: Date/GregorianCalendar use 0-based months.
  - counterexample: String equalsIgnoreCase, compareToIgnoreCase; but regionMatches(boolean ignoreCase).
  - counterexample: String .length(), array .length, collection .size() .

Law of Demeter

- **Law of Demeter**: An object should know as little as possible about the internal structure of other objects with which it interacts.
  - An object, especially an "immutable" one, should not expose its representation by returning a reference to its internal goodies.
    - sometimes called "shallow immutability" if not done properly

- **representation exposure:** When an object allows other code to examine or modify its internal data structures. (A bad thing.)

- If your object has an internal collection:
  - Don't return it! Or return a copy, or an immutable wrapper.
- If your (immutable?) object has mutable objects as fields:
  - Don't let clients access them! Copy them if sent in from outside.

Law of Demeter violation

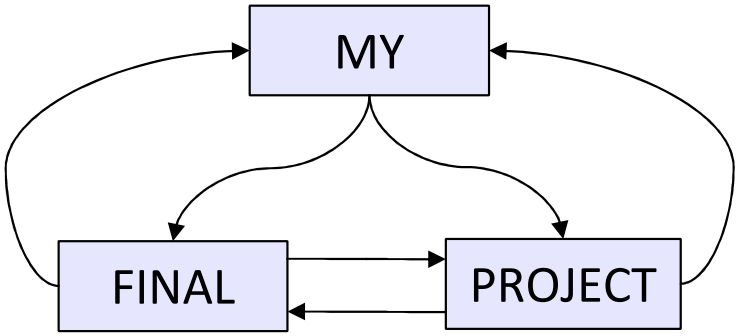
- bad: `general.getColonel().getMajor(m).getCaptain(cap)`
  `.getSergeant(ser).getPrivate(name).digFoxHole();`
  - "inappropriate intimacy": too-tight chain of coupling between classes
- better: `general.superviseFoxHole(m, cap, ser, name);`

- an object should send messages only to the following:
  - 1. itself (this)
  - 2. its instance variables
  - 3. method's parameters
  - 4. any object it creates
  - 5. any object returned by a call to one of this's methods
  - 6. any objects in a collection of the above
    - notably absent: objects returned my messages sent to other objects

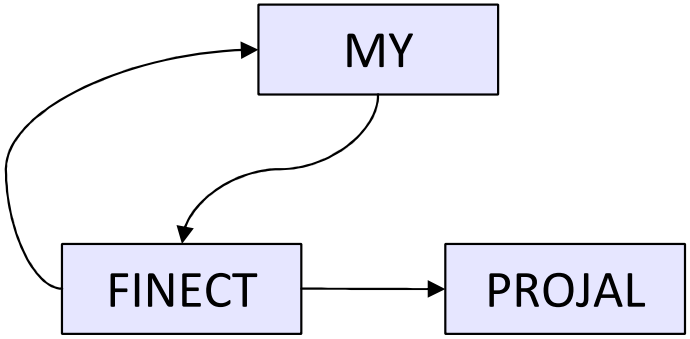
6) Coupling

- **coupling**: Amount of interaction between classes/parts of a system.
  - To simplify, split design into parts that don't interact much.
    - Coupling leads to complexity
    - Complexity leads to confusion
    - Confusion leads to suffering!

An application



A poor decomposition (parts strongly coupled)



A better decomposition (parts weakly coupled)

Invariants

- **class invariant**: An assertion that is true about an object or class throughout its lifetime.
  - e.g. A `BankAccount`'s balance will never be negative.

- Think carefully about what invariants are important for your class.
  - State them in your documentation, and enforce them in your code.

- What invariants are there on the state of these classes?
  - `Time` / `Course` (HW2)
  - `Item` / `Purchase` / `ShoppingCart` (HW1)
  - `ArrayList` / `HashMap`

Documenting a class

- Keep internal and external documentation separate.
  - *external*: /** ... */ Javadoc atop class 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.
  - Missing either of these types of documentation is poor style.

The role of documentation

- Kernigan and Plauger on role of documentation:
  - 1. If a program is incorrect, it matters little what the docs say.
  - 2. If documentation does not agree with code, it is not worth much.
  - 3. 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.
  - 4. Comments should provide additional information from the code itself. They should not echo the code.
  - 5. 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
  - key Q: "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.
  - (while still adhering to the preceding design principles)
  - Clients like classes that are simple to use and understand.

- Achieve a minimal public interface by:
  - Removing unnecessary methods.
  - 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 `enum`s, even with only 2 values.
  - Bad: `oven.setTemp(97, true);  // Celsius`
  - Good: `oven.setTemp(97, Temperature.CELSIUS);`
- Wrapper types should be used minimally (usually with collections).
  - **EJ Tip #49**: Prefer primitive types to boxed primitives.
    - Bad: `public Counter(Character ch)`

View independence

- Confine user interaction to a core set of "view" classes and isolate these from the classes that maintain the key system data.
  - e.g. `ShoppingMain`, `ScheduleGUI`

- 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.
  - Bad: `public void printMyself()`
  - Good: `public String toString()`

Design exercise

- Suppose we are writing a birthday-reminder app and we've decided that it needs the following classes:
  - Date: Represents a particular day on which birthdays can fall.
  - Birthdays: Represents all people whose birthdays I want to remember.

  - What fields do they have?
  - What constructors do they have?
  - What methods do they provide?
    - static?
  - Is there anything we can leave out?
  - What invariants should we guarantee?