Object-oriented Design

Andrew Black

Based on Chapter 1 of POODR

The Problem Design Solves

- Accommodating Change
  - “What’s the most important property that you feel your programs should possess?”
  - “Change requests are the programming equivalent of friction and gravity”

Why “Design”?

- Metz: “Immerse yourself in Objects”
  - everything else is commentary!
  - Sharp’s “Object-oriented thinking”
- Metz: Design is about *managing dependencies*
  - the fewer the dependencies, the more change the software can tolerate.

Design is the Art of arranging code

- Designs that anticipate specific future requirements almost always end badly.
  - Practical design does not anticipate what will happen to your application, it merely accepts that something will and that, in the present, you cannot know what
- The purpose of design is to allow you to do design later
  - its primary goal is to *reduce the cost of change*.
  - It does this by *minimizing dependencies.*

Design & Coding go together

- The problem is not one of technical knowledge but of organization;
  you know how to write the code but not where to put it.
- Don't try to anticipate specific future requirements
  - don't choose: leave room to maneuver

“Principles”

- You may have heard of:
  - SOLID: single responsibility, Open-Closed, Liskov Substitutability, Interface Segregation, and Dependency Inversion
  - DRY: Don't Repeat Yourself
  - Law of Demeter
  - Tell, don't Ask

SOLID (from Wikipedia)

SRP
Single responsibility principle: an object should have only a single responsibility.

OCP
Open/closed principle: "software entities … should be open for extension, but closed for modification".

LSP
Liskov substitution principle: "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program". See also design by contract.

ISP
Interface segregation principle: "many client-specific interfaces are better than one general-purpose interface."

DIP
Dependency inversion principle: one should "Depend upon Abstractions. Do not depend upon concretions." **Dependency injection** is one way of following this principle.

Don't be fooled!

- These are not principles like Archimedes' or Heisenberg's!
- They are *heuristics* that capture what has been shown to be good practice.
  - You can violate the principles
  - But you better know why you did so!

Time-value of Code

Metz (p11):

> "Sometimes the value of having the feature right now is so great that it outweighs any future increase in costs. If lack of a feature will force you out of business today it doesn't matter how much it will cost to deal with the code tomorrow; you must do the best you can in the time you have.

Time-value of Code

"Making this kind of design compromise is like borrowing time from the future and is known as taking on technical debt. This is a loan that will eventually need to be repaid, quite likely with interest."

Agile design *vs.* BUFD

- Agile Design: arranging the code so that it can *Embrace Change*
  - recognizing that you will *never* know less about the code than before you have written any of it.
- BUFD: completely specifying and documenting the future workings of the whole program before you write any of it

Application as Language

- Each OO application gradually becomes a unique programming language that is specifically tailored to your domain.
- Whether this language ultimately brings you pleasure or gives you pain is a matter of design.
  - See also Baniassad & Myers: “An Exploration of Program as Language”

Patterns

- Don’t apply a good pattern to the wrong problem
  - drug abuse?
- Design is progressive discovery based in iteration and feedback
  - avoid: underdesign, overdesign, and separating design from programming

“Design”	“design”
deciding, in advance, how to arrange all the code that will ever be	arranging all the code you have now so that it will be easy to change