Design for Change

Design principle for change: information hiding:

- expose little implementation as possible
- allows you to change hidden details later

Subtype Polymorphism:

- There may be multiple implementations of an interface
- Multiple implementations coexist in the same program
- May not even be distinguishable
- Every object has its own data and behavior

Interface:

- can implement start (defining required methods and classes)
- create class to implement interface

What to test:

- Functional correctness of a method (e.g.,computations, contracts)
- Functional correctness of a class (e.g., class invariants)
- Behavior of a class in a subsystem/multiple subsystems/the entire system
- Behavior when interacting with the world
  - Interacting with files, networks, sensors,
  - Erroneous states
  - Nondeterminism, Parallelism
  - Interaction with users
- Other qualities (performance, robustness, usability, security,)

Unit Tests (Junit good for JAVA):

- Unit tests for small units: functions, classes, subsystems
  - Smallest testable part of a system
  - Test parts before assembling them
  - Intended to catch local bugs
- Typically written by developers
- Many small, fastrunning, independent tests
- Little dependencies on other system parts or environment
- Insufficient but a good starting point
- extra benefits:
  - Documentation (executable specification)
  - Design mechanism (design for testability)

Test cases strategies:

- use specs
- representative cases
- invalid cases
- boundary cond
- think like attacker
- dificult cases

static methods:

- Static methods belong to a class
- global
- Direct dispatch, no subtype polymorphism
- Avoid unless really only a single implementation exists (e.g., Math.min)

Best practices:

- control access
  - fields not accessible from client code
  - methods only accessible in exposed interface
- contracts - agreement between provider and user
  - interface specification
  - functionality and correctness expectations
  - Performance expectations
- Visibility Modifiers
  - design principle for change: information hiding
  - expose little implementation as possible
  - allows you to change hidden details later

Notes:

- try to avoid setters
- Organize program functionality around kinds of abstract objects
  - For each object kind, offer a specific set of operations on the objects
  - Objects are otherwise opaque: Details of representation are hidden
  - Messages to the receiving object
- Distinguish interface from class
  - Interface: expectations
  - Class: delivery on expectations (the implementation)
  - Anonymous class: special Java construct to create objects without explicit classes: Point x = new Point() /* implementation */ ;
- Explicitly represent the taxonomy of object types
  - This is the type hierarchy (!= inheritance, more on that later): A CartesianPoint is a Point
- Design Patterns!!
  - Design Patterns by Gamma,Helm,Johnson,Vlissides

sources:

- `http://www.cs.cmu.edu/~charlie/courses/15-214/2015-fall/index.html#schedule`