Design Principles

aka Object Oriented Programming

Shaun Luttin

September 30, 2017

Goal

- Become vaguely familiar with these principles.
- Have a starting off point for further research.

Why?

- Allow change (of system capacities) without redesign.
- Allow reuse in other applications.

Encapsulate what varies.

- Encapsulate . . .
  - Restrict outside access to a thing's parts.
  - Bundle operations with the things they use.
- . . . what varies.
  - This refers to changes to source code.
  - Source code changes due to changing requirements.
  - Requirements change for a lots of reasons.
  - E.g. A change in government may cause a change in tax law.
- Restrict outside access to parts of the source code that might change due to changing requirements.
- "what [do] you want to be *able* to change without redesign?" (Gamma et al, 1977)

Program to interfaces not to implementations.

- an interface says only what requests it will receive
- an implementation says how it will handle those requests
- programming to interfaces helps because it
  - lets us easily change an implementation, even at runtime
  - allows applications to send the same request to different classes
- Interface Segregation Principle (Martin, 1996)
- SOLID
  - Define an interface that is specific to the needs of the client.
  - "Clients should not be forced to depend upon interfaces that they do not use." (Martin, 1996)

Depend on abstractions not on concrete classes.

- interfaces and abstractions are similar: neither can exist
- concrete classes can exist (i.e. can become objects)
- to depend on something means a direct reference to it
- The Dependency Inversion Principle (Martin, 1996)
- SOLID
  - Traditionally, high-level modules depend on low-level modules:
  - Higher $\rightarrow$ Middle $\rightarrow$ Lower $\rightarrow$ ...
  - Dependency Inversion inverts that:
  - Higher $\rightarrow$ Abstraction $\leftarrow$ Middle $\rightarrow$ Abstraction $\leftarrow$ Lower ...
- When layering, higher-levels define the abstractions
- and lower-levels implement the abstractions.
- Why? Enable reuse of higher-level modules.

Only talk to your friends.

- The Law of Demeter (Holland, 1987)
- aka The Princple of Least Knowledge
- Why? Promotes loose coupling via encapsulation.
- “Only talk to your friends”
- “Only use one dot”
  - More than one dot is cause for reflection;
  - it is not necessarily a violation of the LoD.
  - E.g. fluent interfaces use many dots.

A class should have only one reason to change.

- SOLID
- The Single Responsibility Principle (Martin, 2003)
- "A class should have only one reason to change"
  - Recall from "encapsulate what varies."
  - This refers to changes to source code.
  - Source code changes due to changing requirements.
- Why?
  - (Re)use feature X without bringing feature A-Z.
  - Change feature X without breaking/recompiling what depends on feature A-Z.

Don't call us, we'll call you.

- "Hollywood Principle" (Sweet, 1983)
- "Inversion of Control" (Johnson and Foote, 1988)
  - Dependency injection is a type of Inversion of Control
  - IoC containers are a type of Dependency Injection
- Dependency Inversion - who owns the abstraction?
- Inversion of Control - when do things happen?
- "coordinating and sequencing application activity"
- "makes a framework different from a library":
  - library: "a set of functions you can call"
  - framework: "insert your behavior into various places"
- How? subclassing, implementing iterfaces, binding/events

Classes should be open to extension and closed for modification.

- SOLID
- The Open-Closed Principle
- Once it is shipped, the source code is sacrosanct.
- Rather than change the source code and risk breaking it,
- extend the source code via inheritance or wrapping.
- E.g. the Decorator Pattern (Gamma et al, 1977)

Favour composition over inheritance.

- Composition means a has-a relationship.
  - It is often more semantically natural.
  - It lets us swap implementations at runtime.
- Inheritance means an is-a relationship.
  - Tall class heirachies are brittle.
  - Changing an implementation is limited to compile time.
  - It is harder to do correctly.
- The Liskov Substitution Principle (Liskov and Wing, 1994)
  - SOLID
  - A consumer that is expecting A,
  - should have no surprises on receiving a child of A.
  - Compilers do not help: this is a semantic syntactic contraint.
  - e.g. class Hemlock should probably not inherit class Vegetable.

Strive for loosely coupled designs among objects that interact.

- This is the summary statement for all the princples.
- When loosely coupled, we can . . .
- . . . change X without needing to change Y, and
- . . . use X without needing to bring along Y.