

Design Quality

- There are various approaches to assessing the quality of an object-oriented design
 - Design reviews
 - Metrics
 - Prototyping
 - Adherence to guidelines

Design Guidelines

- A design principle/guideline is an informal piece of advice
- WRT OO, design guidelines take the form of dos and don'ts
 - E.g. bad smells, heuristics
- We will survey some of the more well-known design principles
- But first we will review three important foundational concepts

Coupling

- The extent to which a module is interdependent with other modules
 - Strive for low coupling
- Reduces software maintenance costs
- Constantine

Cohesion

- The extent to which a module has a single purpose
 - Strive for high cohesion
- Code readability and reusability increased
- Constantine

Orthogonality

- Design systems so that features can be varied independently
- Clarifies system descriptions
- May lead to weird combinations

Information Hiding Principle

- *Information hiding ... is the principle of hiding of design decisions ... that are most likely to change, thus protecting other parts of the program from change if the design decision is changed.* -- Wikipedia
- Also called *encapsulation*
- Violated by inheritance
- Parnas

Liskov Substitution Principle

- Subclass instances should satisfy parent-class definition
 - If a client accepts and works correctly on a parent class instance it should also work on a child-class instance
- This implies that child class instances should obey parent class invariants and method contracts (pre/post conditions)
 - Child method preconditions weaker
 - Postconditions stronger
 - "expect no more and provide no less"

Law of Demeter

- *A method M of an object O may only invoke the methods of the following kinds of objects: O itself, M's parameters, any objects created/instantiated within M, O's direct component objects* -- Wikipedia
- Reduces intermodule dependencies
- May lead to wrapper classes
- Lieberherr

Hollywood Principle

- An object-oriented frameworks should make calls to client code rather than being called from it
 - An OO *framework* is a set of collaborating abstract classes
 - "Don't call us; we'll call you"
 - Control inversion
 - Wallace

Dependency Inversion Principle

- *High level modules should not depend upon low level modules. Both should depend upon abstractions* -- Martin
- Related to inversion of control
- Opposite of a traditional layered architecture

Open-Closed Principle

- "A class should be open for extension but closed for modification"
- Don't change a class after it has been delivered (other than bug fixes)
- Two views
 - Implementation inheritance (Meyer)
 - Interface inheritance (Martin)

Interface Segregation Principle

- Have a given client depend on an interface to a part of the large classes features rather than directly on the large class
 - Break complex interfaces into pieces
- Collaboration/Role-based design
- Martin

Release Reuse Equivalency Principle

- "The granule of reuse is the granule of release"
 - Reusers want to track changes
- Packages are a good unit that works well with reuse and release
- Martin

Common Closure Principle

- Classes that change together, belong together
- Minimize dependencies that impact change
- Martin

Common Reuse Principle

- Classes that aren't reused together, should not be grouped together
- Reduce impact on reusers when something changes
- Martin

Acyclic Dependency Principle

- The dependencies between packages must not form cycles
- If A depends on B and B depends on A, invent C, take the part of A that B depends on and place it in C, and have both A and B depend on C
- Another way to break a cycle where the packages are siblings is to add an interface to B (Aint) and have A implement it
- Martin

Stable Dependencies Principle

- "Depend in the direction of stability"
- Reduce the amount of effort that is required to make a change
 - One factor that affects the changeability of a component of a system is its connections to other components
- If a lot of other components depend on a given component, then when the given component changes, a lot of other components may have to be changed
 - In this scenario, Martin says that the given component is "stable", indicating that a lot of effort is required to change it
 - We usually think of "stability" as a positive term, but in this case, Martin is treating it as undesirable
- Martin

Stability Metric

- $I = C_e / (C_a + C_e)$
 - where C_e is efferent coupling (number of components that the given component depends on), and C_a is the afferent coupling (number of components that depend on the given component)
- His metric and his definitions are taken from different points of view. He first defined stability in terms of components above
- But his metric is terms of components below. I am not saying that there is a bug in the metric, only that it is hard to understand because of the switch in point of view
- Also, the fact that he is defining the concept of "stability" of a component, implies that it is a property of the component.
- Actually, it is a property of the *****use***** of the component."
- The dependency graph should fan out as you move downward

Stable Abstraction Principle

- Stable packages should be abstract packages
 - Packages that are depended on by a lot of other packages should be abstract
 - They are hard to change but easy to extend
- **Abstractness = $A = N_a / N_c$**
 - N_a = # of abstract classes
 - N_c = total # of classes
- **Normalized Distance = $D' = | A + I - 1 |$**
 - Lower is better

Bad Smells

- Symptoms of design problems
 - Such as duplicate code, too many comments, long classes
- Recognition part of refactoring
- Beck, popularized by Fowler

Design Heuristics

- Users of a class must be dependent on its public interface, but a class should not be dependent on its users
- Distribute system intelligence horizontally as uniformly as possible
- Do not create god classes/objects in your system
- Most of the methods defined on a class should be using most of the data members most of the time
- Distribute system intelligence vertically down narrow and deep containment hierarchies
- Check constraints in constructors rather than in method preconditions where possible
- Riel

Factoring

- "Factor the commonality of data, behavior, and/or interface as high as possible in the inheritance hierarchy"
- Riel

Inheritance

- "Inheritance should be used only to model a specialization hierarchy"
- Prefer composition/aggregation /delegation over inheritance
- "It should be illegal for a derived class to override a base class method with a NOP method"
- Gosling: throw out class inheritance?
- Riel

Abstraction

- "Do not change the state of an object without going through its public interface"
- Does this mean that a method in a class cannot change an instance variable without calling the setter?
- Riel

Transparency

- A change is transparent if the system after change adheres to previous external interface as much as possible while changing its internal behavior -- Wikipedia
- Shield clients on the other end of the interface
- Refers to invisibility of the component not to visibility of component's internals
- Middleware

Intentionality

- The extent to which the designer's intent is manifest and localized in the code
- Supports traceability, validation, and maintainability

Sources

- Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts.
Refactoring: Improving the Design of Existing Code.
Addison-Wesley, 1999.
- Robert C. Martin.
Clean Code / A Handbook of Agile Software Craftsmanship.
Prentice Hall, 2008.
- Arthur J. Riel.
Object-Oriented Design Heuristics.
Addison-Wesley, 1996.