Patterns in Design Patterns

General design pattern rules.
First Rule

- There are no rules.
  - This is not mathematics (yet).
  - Every situation may need different approach.

- But, over and over, the same principles apply.
  - Guiding principles.
Obey Good OO and Soft. Eng. Precepts

- Modularity
- High cohesion.
- Low connectivity.
- Encapsulation.
  - Hide implementation details and only show interface to the object.

- Enough said.
Encapsulate Change

This is fundamental!

Find what is varying and separate it.

- Why?
  - Lowers connectivity, making it easier to replace.
  - Increases cohesion (if/else), making it easier to replace.

- Examples:
  - Bridge encapsulates change to reduce connectivity.
  - Abstract Factory encapsulates changing classes to increase cohesion.
  - Strategy pattern encapsulates changing methods to increase cohesion.
Favor Composition over Inheritance

1. Inheritance suggests something is changing at the new (child) level.
   - Adds new behavior.
   - Changes existing behavior (polymorphism – overriding methods in child classes).

2. Problem: Should be encapsulating this change!

3. Instead compose (or aggregate).
   - E.g., Bridge pattern.
   - E.g., Strategy pattern
   - Chains of inheritance are overused.
     - Too much of a good thing.
     - We’ll discuss this with anti-patterns – Golden Hammer!
Open-Closed Principle

- Never want to change existing code.
  - Why?
    - Modifications can introduce errors.
    - QA will have you fired!

- Instead find ways to “extend” code without changing existing code.
  - Not necessarily “extend” as in Java inheritance.
    - See “Favor Composition Over Inheritance.”

- Open to extension, closed to modification.
  - E.g., Façade, Adapter, Bridge, Abstract Factory, etc.
    - Can add, reduce, change functionality w/o modifying.
  - Difficult to do if *refactoring* with a pattern that works with methods (“behavioral patterns”). e.g., Observer Pattern.
Design to Interfaces

- What’s an interface? The public methods/variables.

- Create interfaces that are *not* implementation specific.
  - Why?
    - Implementation can vary!
    - E.g., shell sort versus insertion sort.

- Encapsulate implementation variability.
  - Users don’t need to know how it is done (cohesion issues).
    - E.g., do you have any idea what goes on inside the “String” class? No! You just know it works and does its thing.
  - E.g., Strategy pattern hides the variable algorithm (method). User of design’s interface doesn’t know which algorithm is used.

- So build generic interfaces.
  - Use patterns to hide the implementation.
Types of Patterns

- **Structural** – works with classes. Combines groups of classes.
  - Façade
  - Adapter
  - Bridge

- **Creational** – instantiates objects.
  - Abstract Factory
  - Singleton

- **Behavioral** – works with methods. Allows flexible behavior.
  - Strategy
  - Observer
A Pattern of Problems: Behavioral Patterns

Tend to work on methods

- Hard to draw appropriately with UML class diagram.
- Messy because makes requirements on class methods. Can lead to
  - multiple inheritance or implementations
    - what if class already inherits from something?
  - or, restructuring of the class methods (not modular)
  - BUT, sometimes best way to encapsulate variable methods.

MyClass extends YourClass implements TheirClass

e.g., Observer pattern

adds new methods