SOLID

Open / Closed Principle (OCP)

- Classes should be open for extension, but closed for source modification
  - Extending functionality of a class shouldn't require modifying existing code, just adding to it
- Can't extend without changing base class
- Not as bad as in statically typed languages... but still ugly
  - Statically typed language $\triangleq$ abstract factory pattern
  - Ruby has a simple implementation of this with `constantize` method which turns a string into a callable class
- Template method pattern $\triangleq$ set of steps is the same, but implementation of steps is different
  - Inheritance $\triangleq$ subclasses override abstract 'step' methods
- Strategy pattern $\triangleq$ task is the same, but many ways to do it
  - Composition $\triangleq$ component classes implement whole task
- Goal: prefer composition over inheritance
- Decorator Pattern $\triangleq$ class that wraps another class for a certain functionality
- Can't close against all types of changes, so have to choose, and you might be wrong
- Agile methodology can help expose important types of changes early
  - Scenario-driven design with prioritized features
  - Short iterations
  - Test-first development

Liskov Substitution Principle (LSP)

- 'A method that works on an instance of type T should also work on any subtype of T'
- Contracts $\triangleq$ composition over inheritance
  - If you can't express consistent assumptions about 'contract' between class and collaborators, likely LSP violation
- Symptom: change to subclass requires change to superclass
- Treat classes like primitives

Injection of Dependencies (Dependency Injection)

- Problem $\triangleq$ a depends on b, but b's interface and implementation can change even if functionality is stable
- Solution $\triangleq$ 'inject' an abstract interface that a and b depend on
  - If not exact match, use adapter or facade
  - 'inversion' $\triangleq$ now b (and a) depend on interface vs a depending on b
- Ruby equivalent $\triangleq$ extract module to isolate the interface

DI Through Adapter Pattern

- Problem $\triangleq$ client wants to use a 'service'
  - Service generally supports desired operations
  - APIs don't match what client expects / client must interpolate transparently with multiple slightly different services
- Solution $\triangleq$ adapt approach to support both APIs

DI Through Facades

- Provide a simplified interface for an API instead
- Doesn't need to adapt every function of the API

Demeter Principle

- Only talk to your friends... not strangers
- You can call methods on yourself or your own instance variables if applicable
- Idea: do not chain methods
- Mock train-wrecks in tests are a symptom of not following this
- Solutions
  - Replace method with delegate
  - Separate traversal from computation (visitor)
  - Be aware of important events without knowing implementation details (observer)

Delegation

- Feature envy occurs when you're 'reaching into' a related class to often
- Extract methods to isolate the boundary
- Focus on the behavior and delegate that

Observer

- Problem $\triangleq$ entity O (observer) wants to know when certain things happen to entity S (subject)
- Design issue: acting on events is O's concern - don't want to pollute S
- Any type of object could be an observer or subject

More 'Adapter' Patterns

Null Object Pattern

- Problem $\triangleq$ want invariants to simplify design, but app requirements seem to break this
- Null object $\triangleq$ stand-in on which 'important' methods can be called
  - The return values for these methods would correlate with the null case

Singleton Pattern

- Technically a class that provides only 1 instance which anyone can access
- eg: a global var that others can reassign or a constant

Proxy Pattern

- Implements same methods as 'real' service, but 'intercepts' each call
  - Do authentication / protect access
  - Defer work

P&D Perspective on Design Patterns

- Can consider design patterns in design phase
- Critique $\triangleq$ no code until design complete $\implies$ no confidence design implementable; matches customer needs (when start coding, design must change)
- Agile critique $\triangleq$ encourages developers to start coding without any design
  - Relies too much on refactoring later
- Agile advice $\triangleq$ if previously done project that has some design constraints or elements, OK to plan for in similar project, as experience likely leads to reasonable design decisions