Dependency Inversion

Overview

- dependency inversion is a principle that states that entities must depend on abstractions and not on concretions
  - the goal is to reduce dependencies on concrete classes
- abstractions should not depend upon details
  - details should depend upon abstractions
- high level classes must not depend on the low level classes
  - both high-level classes and low-level classes should depend upon abstractions
  - the lower-level class implementation is accessible to the higher-level class via an abstract interface
  - actual implementation of lower level class can then vary
- the "inversion" in the name "Dependency Inversion Principle" is there because it inverts the way you typically might think about your OO design
  - top-to-bottom dependency has inverted itself, with both high-level and low-level classes now depending on an abstraction
- sounds a lot like "Program to an interface, not an implementation"
  - similar, however, the Dependency Injection Principle makes an even stronger statement about abstraction
- dependency inversion is a central principle underlying the use of design patterns

Invert your thinking...

- lets say we need to implement a pizza store
  - What's the first thought that pops into your head?
- start at the top and follow things down to the concrete classes
  - however, you do not want your store to know about the concrete pizza types
  - pizza store will then be dependent on all those concrete classes
- let's "invert" your thinking...
  - instead of starting at the top, start at the Pizzas and think about what you can abstract
    - pizza is the abstraction
- your different concrete pizza types depend only on an abstraction and so does your store
  - the initial design where the store depended on concrete classes can be inverted to have the design abstract those dependencies

PizzaStore (Example)

- a PizzaStore could be a high-level class
  - its behavior is defined in terms of pizzas
  - it creates all the different pizza objects
  - It prepares, bakes, cuts, and boxes pizzas

- the pizzas it uses are low-level classes
  - pizza implementations are our "low-level classes"
    - VeggiePizza
    - NYStyle
    - ChicagoStyle

- the PizzaStore class is dependent on the concrete pizza classes

- this principle tells us we should write our code so that we are depending on abstractions, not concrete classes
  - applies to both our high-level classes and our low-level classes
  - we can create an abstract class named Pizza
  - the PizzaStore and the concrete pizzas both depend on the Pizza class (the abstraction)

Advantages of Dependency Inversion

- removes tight coupling that comes with a top-down design approach
  - each higher level class is tightly coupled with its lower level concrete class
    - any change in the lower level class will have a ripple effect in the next higher level class
    - makes it extremely difficult and costly to maintain and extend the functionality of the layers

- Dependency Inversion Principle introduces a layer of abstraction between each higher level class and lower level concrete class
  - higher-level classes depend only on a common abstraction
  - lower-level classes can then be modified or extended without the fear of disturbing higher-level classes
    - as long as it obeys the contract of the abstract interface

- dependency inversion provides loose coupling between higher and lower level classes by introducing an abstraction layer
  - highly beneficial for maintaining and extending the overall system

OO guidelines for adhering to DIP

- no variable should hold a reference to a concrete class
  - use the factory design pattern to avoid this

- no class should subclass from a concrete class
  - If you subclass from a concrete class, you are depending on a concrete class
  - subclass from an abstraction (an interface or an abstract class)

- no method should override an implemented method of any of its base classes
  - If you override an implemented method, then your base class was not really an abstraction to start with
  - methods implemented in the base class are meant to be shared by all your subclasses

- this is a guideline you should strive for, rather than a rule you should follow all the time
  - if you have a class that is not likely to change, and you know it, then it is ok to instantiate a concrete class
  - we instantiate String objects all the time and this violates the principle
    - however, the String class is very unlikely to change

- you should internalize these guidelines and have them in the back of your mind when you design