Todd Vorisek
CS 372
May 02, 2018
Final

Part B

Design pattern synopsis:

Interpreter

Implements a grammar.

Chain of Responsibility

Encapsulates processing elements into a "pipeline" abstraction.

Composite

Useful for implementing a branch/leaf hierarchy.

Visitor

Useful for traversing a branch/leaf hierarchy.

Template

Good for when you want to reduce "boiler plate" or repeated code, and can break task specific code off into smaller subclasses.

Observer

Helps notify a program when changes are made. Useful in event driven programs.

Decorator

A decorator is a pattern that enables an object to have properties appended to it arbitrarily. The example in class was Armor that could have properties such as rusty, or iron applied to it. The item being

Command

Allows a programmer to encapsulate functionality inside of an object. Good for segregating undo, redo, save and other common menu operations logically.

Null Object

Useful for a pretty wide array of tasks from testing to command termination. So much so that some languages have these built in. Wide variety of use cases.

Adapter

Wrap a class inside an adapter class to convert it's interface to something your existing infrastructure can communicate with.

Facade

Provides a unified interface to a group of functions, can simplify interfaces considerably.

State

Implements a state machine.

Mock

Useful for running tests on non-sensitive data, or without using too many system resources.

Part C

Solid design principle for object oriented programming.

S - Single Responsibility Principle

Each class should have only one job or responsibility. This results in small classes that can be easily understood. When designing classes, try to determine their interfaces and when adding functionality, always ask if it makes sense for this class to be handling the underlying code. When refactoring, try to determine if parts of a class could be broken off into a smaller class.

O - Open/Closed Principle

Each class should be open for extension but closed for modification. Interfaces should be provided such that desired functionality can be extended into a class, but member variables should be private wherever possible to insure robust security and existing program correctness. This also helps insure you're abiding the next principle.

L - Liskov Substitution Principle

Objects should be replaceable with subtypes or inherited classes without altering the correctness of the program. Extensions should not modify existing logic in their base classes, except in the case of abstract classes. It's better to have a bunch of classes that each inherit from the same base, than to have a chain of classes where each inherits from the the previous one.

I - Interface Segregation Principle

Interfaces should be client specific, with specific use cases in mind, instead of a more generalized interface. This is intended to keep classes decoupled and independent, enabling easy refactoring, changes, and deployment. ISP is often an implemented as an emergent behavior of the single responsibility principle, mentioned above.

D - Dependency Inversion Principle

Interfaces and instantiations should rely on abstractions rather than exact functionality. This enables decoupling of modules and, if the Open/Closed principle has been observed, easy extension of existing program functionality.