Software Craftsmanship - OOP/D Techniques

1. Craftsmanship Manifesto

Not only working software, but also well-crafted software

Not only responding to change, but also steadily adding value

Not only individuals and interactions, but also a community of professionals

Not only customer collaboration, but also productive partnerships

1.1. Craftsmanship Methodologies

1.1.1. TDD

WIP

1.1.2. BDD

WIP

1.1.3. Pair Programming

WIP

1.1.4. Mob Programming

WIP

2. GRASP

The acronym stands for General Responsibility Assignment Software Patterns (yes, please use GRASP), it groups nine design principles for assigning responsibility to objects.

For each principle below, an associated question or a statement is synthetized from the book to understand how to apply the principle:

1. Expert (Information Expert):
    - What object (or group of objects) has the information needed to fulfill a certain responsibility?
2. Creator:
    - Based on the domain model what object B should be responsible for creating object A?
3. Controller:
    - Object with the responsibility to route UI requests to specific business logic.
4. Indirection:
    - Object(s) with the responsability to maintain low coupling between components by routing the association.
5. Low coupling:
    - Assign responsibility to objects in a way that minimize coupling between them.
6. High cohesion:
    - To what degree are my object operations functionally related, do all operations in my object belong together?
7. Polymorphism:
    - Avoid conditional programming when designing varying behavior by types, leverage polymorphic operations instead.
8. Protected variations:
    - How can I design my objects so that I reduce the impact between them in case of variation?
9. Pure Fabrication:
    - An object that is not representing a real word problem domain and with the only responsibility to support high cohesion and low coupling.

3. GoF Patterns

GoF are documented in the famous Design Patterns: Elements of Reusable Object Oriented Software (by Erich Gamma, Richard Helm, Ralph Jonhson, John Vlissides)

4. GoF Patterns - Creational

- Abstract Factory:

Use this pattern when you want to provide a mechanism to create families of objects without specifying their concrete classes.

- Builder:

The idea behind this pattern it to provide a mechanism to allow create of an object based on different representation. Usually if you have in your design a class with multiples constructors having different types of parameters, then this class is a candidate for a Builder pattern.

- Factory method:

This pattern allows you to defer instantiation of a class to subclasses. This is better achieved through an interface.

- Prototype:

Create multiple objects based on an existing prototype instance. This pattern is better used when an object creation has a significant cost or most of the representation is already known ahead of the need of creation.

- Singleton:

This pattern ensure that a class has one and only one instance available. The pattern must provide a way to access this single instance.

5. GoF Patterns - Structural

- Adapter

Use this pattern when you have a class that needs to be used by a client, but that client expects a different or a specific interface. You can adapt an existing class to be used by another class where the initial type would not be acceptable.

- Bridge

Better used when you want to decouple an abstraction from its implementation. Think about when a client requires a specific abstraction with multiple possible implementations. The client is in charge of specifying the abstraction to be used while providing a bridge to another abstraction for which multiple implementations are possible.

- Composite

Think about this pattern when you want to apply to a different type of classes the same function or use them as if they were the same type of objects. A client class would be able to work with different type of objects at runtime uniformly, while these objects are different.

- Decorator

Use this pattern when you want to add more responsibility to existing objects at runtime without modifying its existing interface. The goal of this pattern is to enhance the parent object behavior by wrapping it with the same interface but provide an enhanced behavior.

- Facade

This pattern is helpful when you want to provide a higher-level interface to a complex subsystem. The facade provides a simple and clear access to a group of classes based on the expected behavior scope.

- Flyweight

Use this pattern when your objects representation is heavy in memory and where you can factor some of the state into separate classes to be referenced by the main objects. The idea behind this pattern is that repeating information can be factored into separate classes and referenced to make our instantiated objects less heavy.

- Proxy

Think of this pattern as an adapter with the same interface, the idea to wrap a service operation around another object that can control the behavior, for example lazy loading, calling remote operation and what not.

6. GoF Patterns - Behavioral

- Chain of responsibility

Think about this pattern when you want to the chain the processing of a request across multiples objects. Basically, pass the request across a chain of multiples objects to allow each to handle the request.

- Command

Use this pattern when a client needs to issue different type of requests. This pattern help you avoid having a large inheritance hierarchy and increase your separation of concerns between the request and the implementation. This pattern allows you also to implement undo operation by saving the state in order to reverse the operation.

- Interpreter
- Iterator

This pattern allows you to provide an access to elements of an aggregate in a predefined order without exposing the implementation.

- Mediator

When you need to increase loose coupling and create a boundary during the interactions between objects, this pattern comes at help by keeping objects from refering to each other explicitly.

- Memento

When you think about this pattern, think about the undo mechanism (CTRL-Z). This pattern allow you to capture and externalize the state of an object in order to be able to restore it when needed.

- Observer

This pattern help you implement the notification mechanism. Ensure that multiples objects are notified when on object is modified.

- State

Use this pattern when you want to change an object behavior during runtime and when its state change. You can think of an object being Locked or Unlocked, depending if a user have clicked on a certain button.

- Strategy

Lets imagine you have different algorithm to do a particular operation. This pattern help you select the right approach upon need, basically, vary the implementation based on strategy upon your need.

- Template Method

Use this pattern whenever you want to provide a partial implementation of an algorithm and defer to subclass the rest of the implementation that is specific to the subclass.

- Visitor

This pattern come at help when you want to apply a certain behavior on an object without having to integrate it into the structure of the class. The pattern let you visit an object in a collection and apply a certain behavior.

7. SOLID

The acronym is composed of the first letter of each principle that is described as follow:

1. Single Responsibility (SRP):
    - A class should have one and only one reason to change.
    - Higher cohesion within a class and clear responsibility.
2. Open Closed (OCP):
    - A class should be open for extension but closed for modification.
    - Make sure that your classes don't require modification each time you want to add capabilities.
3. Liskov Substitution (LSP):
    - Be careful with inheritance, a subclass must be able to replace a parent class without impact on the implementation.
4. Interface Segregation (ISP):
    - Make sure your interfaces are as small as possible, high cohesion between the interface methods.
    - If your implementation is forced to implement an interface they don't use then you need to split your interface into a smaller one.
5. Dependency Inversion:
    - When one module depends on another, its better that the dependency is managed through abstractions.
    - The higher modules should always hold the dependency abstraction.

Each pattern will be described using the ptolemy project or other teams examples. A reference will be pointing to the specific application classes implementing the example.


@nabil
Eternal Coder