Patterns in Software Engineering

Lecturer: Raman Ramsin

Lecture 6

OO Principles as Patterns: GRASP

Open Closed Principle (OCP)

- *Classes should be open for extension but closed for modification.*
- OCP states that we should be able to add new features to our system without having to modify our set of preexisting classes.
- One of the tenets of OCP is to reduce the coupling between classes to the abstract level.
  - Instead of creating relationships between two concrete classes, we create relationships between a concrete class and an abstract class or an interface.

Liskov Substitution Principle (LSP)

- *Subclasses should be substitutable for their base classes.*

- Also called the substitutability principle.

Dependency Inversion Principle (DIP)

- *Depend upon abstractions. Do not depend upon concretions.*
- The Dependency Inversion Principle (DIP) formalizes the concept of abstract coupling and clearly states that we should couple at the abstract level, not at the concrete level.
- DIP tells us how we can adhere to OCP.

Interface Segregation Principle (ISP)

- *Many specific interfaces are better than a single, general interface.*

- Any interface we define should be highly cohesive.

Composite Reuse Principle (CRP)

- *Favor polymorphic composition of objects over inheritance.*

- One of the most catastrophic mistakes that contribute to the demise of an object-oriented system is to use inheritance as the primary reuse mechanism.

- Delegation can be a better alternative to Inheritance.

Principle of Least Knowledge (PLK)

- *For an operation O on a class C, only operations on the following objects should be called: itself, its parameters, objects it creates, or its contained instance objects.*
- Also called the **Law of Demeter**.
- The basic idea is to avoid calling any methods on an object where the reference to that object is obtained by calling a method on another object (*Transitive Visibility*).
- The primary benefit is that the calling method doesn't need to understand the structural makeup of the object it's invoking methods upon.

GRASP

- Acronym stands for General Responsibility Assignment Software Patterns.
- A set of 9 Patterns introduced as a learning aid by Craig Larman.
- Describe fundamental principles for object-oriented design and responsibility assignment, expressed as patterns.
- The skillful assignment of responsibilities is extremely important in object design.
- Determining the assignment of responsibilities often occurs during the creation of interaction diagrams, and certainly during programming.

GRASP: Patterns

1. Information Expert
2. Creator
3. Low Coupling
4. High Cohesion
5. Controller
6. Polymorphism
7. Indirection
8. Pure Fabrication
9. Protected Variations

GRASP: Information Expert

- As a general principle of assigning responsibilities to objects, assign a responsibility to the information expert:
  - i.e. the class that has the *information* necessary to fulfill the responsibility.

GRASP: Creator

- Assign class B the responsibility to create an instance of class A if one or more of the following is true:
  - B *aggregates* A objects.
  - B *contains* A objects.
  - B *records* instances of A objects.
  - B *closely uses* A objects.
  - B *has the initializing data* that will be passed to A when it is created (thus B is an Expert with respect to creating A).
- B is a *creator* of A objects.
- If more than one option applies, prefer a class B which *aggregates* or *contains* class A.

GRASP: Low Coupling

- Assign a responsibility so that coupling remains low.
  - A class with high (or strong) coupling relies on many other classes. Such classes may be undesirable; some suffer from the following problems:
    - Changes in related classes force local changes.
    - Harder to understand in isolation.
    - Harder to reuse because its use requires the additional presence of the classes on which it is dependent.

GRASP: High Cohesion

- Assign a responsibility so that cohesion remains high.
  - A class with low cohesion does many unrelated things, or does too much work.
  - Such classes are undesirable; they suffer from the following problems:
    - hard to comprehend
    - hard to reuse
    - hard to maintain
    - Delicate: constantly affected by change

GRASP: Controller

- Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:
  - Represents the overall system, device, or subsystem *(facade controller).*
  - Represents a use case scenario within which the system event occurs (a use-case- or session-controller).
    - Use the same controller class for all system events in the same use case scenario.
    - Informally, a session is an instance of a conversation with an actor.
- Note that "window," "applet," "widget," "view," and "document" classes are not on this list.
  - Such classes should *not* fulfill the tasks associated with system events, they typically delegate these events to a controller.

GRASP: Polymorphism

- When related alternatives or behaviors vary by type (class), assign responsibility for the behavior — using polymorphic operations — to the types for which the behavior varies.

- Define the behavior in a common base class or, preferably, in an interface.

GRASP: Indirection

- Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled.
- The intermediary creates an *indirection* between the other components.
- Beware of transitive visibility.

GRASP: Pure Fabrication

- Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept — something made up, to support high cohesion, low coupling, and reuse.

- Example: a class that is solely responsible for saving objects in some kind of persistent storage medium, such as a relational database; call it the *PersistentStorage.*
  - This class is a Pure Fabrication — a figment of the imagination.

GRASP: Protected Variations

- Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

- Note: The term "interface" is used in the broadest sense of an access view; it does not literally only mean something like a Java or COM interface.

Design by Contract (DBC)

- A discipline of analysis, design, implementation, and management.
- Bertrand Meyer introduced **Design by Contract** in Eiffel as a powerful technique for producing reliable software.
- Its key elements are assertions: boolean expressions that define correct program states at arbitrary locations in the code.

Design by Contract (DBC): *Assertions*

- Simple assertions belong to individual statements in a program, yet important DBC assertions are defined on classes:
  - Per- method assertions:
    - **Precondition:** A condition that must be true of the parameters of a method and/or data members – if the method is to behave correctly – PRIOR to running the code in the method.
    - **Postcondition:** A condition that is true AFTER running the code in a method.
  - Per-class assertions:
    - **Class Invariant:** A condition that is true BEFORE and AFTER running the code in a method (except constructors and destructors).

Design by Contract (DBC): *Contract*

- Assertions in a class specify a contract between its instances and their clients.
- According to the contract a server promises to return results satisfying the postconditions if a client requests available services and provides input data satisfying the preconditions.
- Invariants must be satisfied before and after each service.
- Neither party to the contract (the clients/consumers and servers/suppliers) shall be allowed to rely on something else than that which is explicitly stated in the contract.

Design by Contract (DBC): *Inheritance*

- The class invariants of all ancestors must also be obeyed by all children; they can also be *strengthened*.
- Inherited preconditions may only be *weakened*. This means a new implementation may never restrict the circumstances under which a client call is valid beyond what has been specified by the ancestor.
  - It may, however, relax the rules and also accept calls that would have been rejected by the ancestor.
- Inherited postconditions may only be *strengthened*. This means a new implementation must fulfil what the ancestors have undertaken to do, but may deliver more.
- "Children should not provide less or expect more than their ancestors."

Reference

- Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development,* 3rd Ed. Prentice-Hall, 2004.