GRASP (object-oriented design)

General Responsibility Assignment Software Patterns (or **Principles**), abbreviated **GRASP**, is a set of "nine fundamental principles in object design and responsibility assignment"[1]:6 first published by Craig Larman in his 1997 book *Applying UML and Patterns*.

The different patterns and principles used in GRASP are controller, creator, indirection, information expert, low coupling, high cohesion, polymorphism, protected variations, and pure fabrication.[2] All these patterns solve some software problems common to many software development projects. These techniques have not been invented to create new ways of working, but to better document and standardize old, tried-and-tested programming principles in object-oriented design.

Larman states that "the critical design tool for software development is a mind well educated in design principles. It is not UML or any other technology."[3]:272 Thus, the GRASP principles are really a mental toolset, a learning aid to help in the design of object-oriented software.

Patterns

In object-oriented design, a pattern is a named description of a problem and solution that can be applied in new contexts; ideally, a pattern advises us on how to apply its solution in varying circumstances and considers the forces and trade-offs. Many patterns, given a specific category of problem, guide the assignment of responsibilities to objects.

Information expert

Problem: What is a basic principle by which to assign responsibilities to objects?
Solution: Assign responsibility to the class that has the information needed to fulfill it.

Information expert (also **expert** or the **expert principle**) is a principle used to determine where to delegate responsibilities such as methods, computed fields, and so on.

Using the principle of information expert, a general approach to assigning responsibilities is to look at a given responsibility, determine the information needed to fulfill it, and then determine where that information is stored.

This will lead to placing the responsibility on the class with the most information required to fulfill it.[3]:17:11

Related Pattern or Principle: Low Coupling, High Cohesion

Creator

The creation of objects is one of the most common activities in an object-oriented system. Which class is responsible for creating objects is a fundamental property of the relationship between objects of particular classes.

Problem: Who creates object `A`?
Solution: In general, Assign class `B` the responsibility to create object `A` if one, or preferably more, of the following apply:

- Instances of B contain or compositely aggregate instances of A
- Instances of B record instances of A
- Instances of B closely use instances of A
- Instances of B have the initializing information for instances of A and pass it on creation.[3]: 16:16.7

Related Pattern or Principle: Low Coupling, Factory pattern

Controller

The **controller** pattern assigns the responsibility of dealing with system events to a non-UI class that represents the overall system or a use case scenario. A controller object is a non-user interface object responsible for receiving or handling a system event.

Problem: Who should be responsible for handling an input system event?
Solution: A *use case controller* should be used to deal with *all* system events of a use case, and may be used for more than one use case. For instance, for the use cases *Create User* and *Delete User*, one can have a single class called *UserController*, instead of two separate use case controllers. Alternatively a *facade controller* would be used; this applies when the object with responsibility for handling the event represents the overall system or a root object.

The controller is defined as the first object beyond the UI layer that receives and coordinates ("controls") a system operation. The controller should delegate the work that needs to be done to other objects; it coordinates or controls the activity. It should not do much work itself. The GRASP Controller can be thought of as being a part of the application/service layer[4] (assuming that the application has made an explicit distinction between the application/service layer and the domain layer) in an object-oriented system with common layers in an information system logical architecture.

Related Pattern or Principle: Command, Facade, Layers, Pure Fabrication

Indirection

The indirection pattern supports low coupling and reuses potential between two elements by assigning the responsibility of mediation between them to an intermediate object. An example of this is the introduction of a controller component for mediation between data (model) and its representation (view) in the model-view-controller pattern. This ensures that coupling between them remains low.

Problem: Where to assign responsibility, to avoid direct coupling between two (or more) things? How to de-couple objects so that low coupling is supported and reuse potential remains higher?

Solution: 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.

Low coupling

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. **Low coupling** is an evaluative pattern that dictates how to assign responsibilities for the following benefits:

- lower dependency between the classes,
- change in one class having a lower impact on other classes,
- higher reuse potential.

High cohesion

High cohesion is an evaluative pattern that attempts to keep objects appropriately focused, manageable and understandable. High cohesion is generally used in support of low coupling. High cohesion means that the responsibilities of a given set of elements are strongly related and highly focused on a rather specific topic. Breaking programs into classes and subsystems, if correctly done, is an example of activities that increase the cohesive properties of named classes and subsystems. Alternatively, low cohesion is a situation in which a set of elements, of e.g., a subsystem, has too many unrelated responsibilities. Subsystems with low cohesion between their constituent elements often suffer from being hard to comprehend, reuse, maintain and change as a whole.[3]:314–315

Polymorphism

According to the **polymorphism** principle, responsibility for defining the variation of behaviors based on type is assigned to the type for which this variation happens. This is achieved using polymorphic operations. The user of the type should use polymorphic operations instead of explicit branching based on type.

Problem: How to handle alternatives based on type? How to create pluggable software components?
Solution: 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. (Polymorphism has several related meanings. In this context, it means "giving the same name to services in different objects".)

Protected variations

The **protected variations** pattern protects elements from the variations on other elements (objects, systems, subsystems) by wrapping the focus of instability with an interface and using polymorphism to create various implementations of this interface.

Problem: How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
Solution: Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

Pure fabrication

A **pure fabrication** is a class that does not represent a concept in the problem domain, specially made up to achieve low coupling, high cohesion, and the reuse potential thereof derived (when a solution presented by the *information expert* pattern does not). This kind of class is called a "service" in domain-driven design.

Related Patterns and Principles • Low Coupling. • High Cohesion.