SOFTWARE DESIGN RESPONSIBILITIES AND QUALITY CONCEPTS

1. Key Points from GRASP Patterns to Assign Responsibilities in Designs

GRASP (General Responsibility Assignment Software Patterns) provides a set of guidelines that help software designers assign responsibilities effectively to objects in object-oriented design. The patterns promote clarity, maintainability, and reusability in software systems by systematically assigning roles to classes and objects. The following are the key GRASP patterns that play a critical role in assigning responsibilities:

1.1 Information Expert

The principle of Information Expert suggests assigning responsibility to the class that has the necessary information to fulfill it. This leads to high cohesion and supports encapsulation, making the system easier to maintain. For instance, if we want to compute the total sale of an order, we should assign this responsibility to the class that holds the line items and prices, typically the Order class.

1.2 Creator

The Creator pattern advises assigning the responsibility of creating an instance of a class to the class that aggregates, contains, or closely uses the created class. This promotes low coupling and logical relationships. For example, an Order class might create OrderLine instances because it contains them.

1.3 Controller

The Controller pattern involves assigning responsibility for handling system events to a controller class that represents a use-case scenario or a façade to the system. It promotes a clean separation between the user interface and business logic.

1.4 Low Coupling and High Cohesion

GRASP promotes low coupling by minimizing dependencies between classes, thus enhancing flexibility and reducing the ripple effect of changes. It also emphasizes high cohesion to ensure each class has a single, well-defined purpose.

1.5 Polymorphism and Indirection

Polymorphism supports responsibility assignment through dynamic behavior based on type. Indirection introduces an intermediate class to mediate between other classes, promoting flexibility and decoupling.

By applying GRASP patterns, developers can build robust, maintainable systems that adhere to good design principles and software engineering practices (Larman, 2004).

2. Coupling and Cohesion

Coupling and cohesion are two core concepts in software engineering that directly impact a system's maintainability and readability.

2.1 Coupling

Coupling is the degree of reliance between modules of software. A high coupling is that the module depends a lot on others, and low means that the module can operate independently. Low coupling is good because it means the system is more modular and tweaks without affecting the other parts of the system.

For instance, a User module guaranteeing lots of Book module implementation details would be tightly coupled, in a library management system. This means that if you change something in the Book module, it may affect the User module.

2.2 Cohesion

Cohesion: This is a measure of relative relatedness of the responsibilities of a single module or class. High cohesion is when the members of a class/module cooperate tightly, so one module does one focused thing, which makes it clearer and more maintainable. Low cohesion means the module has a set of unrelated responsibilities which makes it difficult to comprehend and maintain.

For an example relating to the library system, a class BookManager whose only responsibility is managing the book inventory, is said to be highly cohesive. A class which deals with book inventory, user logins, and overdue fines would not have cohesion and would become unmaintainable.

2.3 Ideal Combination: High Cohesion and Low Coupling

High cohesion and low coupling are the best design goals. This combo makes sure that each module is focused and self-sufficient (high cohesion) and yet is loosely dependent on others (low coupling). This architecture contributes to better scalability, testability, and reliability in the system.

For instance, consider a system where a PaymentProcessor class handles only payment logic and interacts with an abstract PaymentGateway interface. The class has high cohesion (focused responsibility) and low coupling (depending on an interface, not specific implementations). This setup allows developers to change the payment gateway without altering the processor logic, adhering to the Open/Closed Principle of software design.

Studies from software engineering also support this strategy. Pressman and Maxim (2015) described that, high cohesion and low coupling of a Software reduces its system complexity and improves flexibility for changes. Further, the point is made by Sommerville (2016) that, as modular systems, they are easier to understand, test, and reuse.

Wordcount: 686

References:

Aldrich, J.I., & Garrod, C. (2015). Assigning Responsibilities [Presentation slides]. In *Principles of Software construction: Objects, design and concurrency.* Institute for Software Research. https://www.cs.cmu.edu/~charlie/courses/15-214/2015-fall/slides/03b-assigning-responsibilities.pdf

Larman, C. (2005). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall. https://www.amazon.com/Applying-UML-Patterns-Introduction-Object-Oriented/dp/0131489062

Pressman, R. S., & Maxim, B. R. (2014). *Software engineering: A practitioner's approach* (8th ed.). McGraw-Hill Education. https://www.amazon.com/Software-Engineering-Practitioners-Roger-Pressman/dp/0078022126

Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson. https://www.pearson.com/en-us/subject-catalog/p/software-engineering/P200000003258/9780137503148