GRASP: (General Responsibility Assignment Software Patterns)

1. Information Expert

Assign the responsibility for a task to the class that has the necessary information to fulfill it.

Example:

```
class Customer {
        private String name;
        private double balance;

        public double calculateBalance() {
                return this.balance;
        }
}
```

2. Creation Expert

Assign the responsibility of creating an instance of a class to the class that has the information necessary to create that object or is closely related to it.

Means: A class should be responsible for creating instance of objects that are closely related to it.

Example:

```
class Order {
        private List<Item> items = new ArrayList<>();

        public void addItem(Item item)  {
                items.add(item);
        }

        public Item createItem(String name, double price) {
                return new Item(name, price);
        }
}
```

3. Controller

Assign the responsibility of handling the system events to a class that represents a use-case scenario or a system operation.

Example:

```
class BookController {
    private BookStore store;

    public void addBook(String title, String author) {
        store.addBook(new Book(title, author));
    }

    public void removeBook(Book book) {
        store.removeBook(book);
    }
}
```

4. Low Coupling:

Assign responsibilities in such a way that classes and objects have low dependencies on one another.

Example:

```
class Product {
    private String name;
    private double price;
}
class Inventory {
    private List<Product> products = new ArrayList<>();

    public void addProduct(Product product) {
        products.add(product);
    }
}
```

If you have a Inventory class, the Product class should not directly reference the Inventory class. Instead Inventory class should handle adding/removing products independently.

5. High Cohesion:

Assign responsibilities to class with high internal cohesion - meaning the class has closely related tasks.

Example:

```
class Customer {
        private String name;
        private String address;

        public void placeOrder(Order order) {
                // Logic for placing an order.
        }
}
```

A Customer class should be focused on attributes and behaviours related to Customers, such as names, address, and methods like placeOrder().

It should not handle payment processing or other unrelated concerns.

6. Polymorphism:

Assign responsibilities for behaviour that can vary based on the type of the objects to the subclass that implements it.

Use polymorphism to handle variations in behaviour.

Example:

```
class Shape {
        public void draw() {}
}
class Circle extends Shape {
        public void draw() {}
}
class Rectangle extends Shape {
        public void draw() {}
}
```

7. Pure Fabrication:

Assign responsibilities to classes that do not represent a real-world concept but help in the design (e.g., utility or helper classes)

Example:

```
class DatabaseConnection {
    public void connect() {
        // Code to establish DB connection
    }
}
```

8. Indirection:

Assign responsibilities to an intermediary object that can mediate between other objects or classes.

Example:

```
class PaymentService {
    private PaymentProcessor paymentProcessor;

    public void processPayment(Customer customer, double amount) {
        paymentProcessor.process(customer, amount);
    }
}
class PaymentProcessor {
    public void process (Customer customer, double amount) {
        // Logic to process payment
    }
}
```

9. Protected Variations:

Assign responsibilities to a class or module that is subject to change, but ensure that the changes don't affect other parts of the system.

Example:

```
interface PaymentMethod {
    void pay(double amount);
}

class CreditCardPayment implements PaymentMethod {
    @Override
    public void pay(double amount) {
```

```
            // Credit card payment logic.
        }
    }
    class PaypalPayment implements PaymentMethod {
        @Override
        public void pay(double amount) {
            // Paypal payment logic.
        }
    }
```

Summary:

1. Information Expert: Assign tasks to the class with the relevant data.
2. Creator: Assign creation responsibility to the class that has necessary information.
3. Controller: Assign handling of events or system actions to a controller class.
4. Low Coupling: Keep classes independent and minimize dependencies.
5. High Cohesion: Keep related tasks together in the same class.
6. Polymorphism: Delegate behaviour variation to subclasses.
7. Pure Fabrication: Use helper classes for technical purposes.
8. Indirection: Use an intermediary to decouple direct interaction between objects.
9. Protected Variations: Protect the system from changes by using abstractions.