UTCN
Computer Science Department
Software Design 2011/2012

GRASP – General Responsibility Assignment Software Patterns

==

I. Theoretical Background

This section briefly presents the basic concepts regarding the General Responsibility Assignment Software Patterns. The provided information is based on [1].

To arrive at a good object-oriented design we use principles applied during the creation of interaction diagrams and/or responsibility assignment: e.g. GRASP patterns.

The UML defines a responsibility as "a contract or obligation of a classifier". Responsibilities are related to the obligations of an object in terms of its behavior. Basically, these responsibilities are of the following two types: *knowing* and *doing*.

- Knowing responsibilities of an object include:
  - knowing about private encapsulated data;
  - knowing about related objects;
  - knowing about things it can derive or calculate;
- Doing responsibilities of an object include:
  - doing something itself;
  - initiating action in other objects;
  - controlling and coordinating activities in other objects;
- A responsibility is not the same as a method, but methods are implemented to fulfill responsibilities. Responsibilities are implemented using methods that either act alone or collaborate with other methods and objects.

Patterns:

- **Expert**
- **Creator**
- **Controller**
- Low Coupling
- High Cohesion
- Polymorphism
- Pure Fabrication
- Indirection
- **Don't Talk to Strangers (Law of Demeter)**

1. Expert

Problem: What is the most basic principle by which responsibilities are assigned in object-oriented design?

Solution: Assign a responsibility to the information expert - the class that has the information necessary to fulfill the responsibility.

Discussion:

- Expert is used more than any other pattern in the assignment of responsibilities.
- Whenever information is spread across different objects, they will need to interact via messages in order to share the work.
- A general advice is to start assigning responsibilities by clearly stating the responsibility!!!
    - Example: "Who should be responsible for knowing the grand total of a sale in a POS application?"
        - By Expert, we should look for that class of objects that has the information needed to determine the total.

Advantages:

- Information encapsulation is maintained since objects use their own information to fulfill tasks => supports low coupling
- Behavior is distributed across the classes that have the required information => more cohesive "lightweight" class definitions

2. Creator

Problem: Who should be responsible for creating a new instance of some class?

Solution: Assign class B the responsibility to create an instance of class A if one of the following is true:

- B contains A objects
- B closely uses A objects
- B has the initializing data that will be passed to A when it is created.

Discussion:

- Creator guides assigning responsibilities related to the creation of objects.
- Sometimes a creator is found by looking for the class that has the initializing data that will be used during creation.

Advantages: Low Coupling is supported

Disadvantages: Complex creation procedures

3. Controller

Problem: Who should be responsible for handling a system event?

- A system event is a high level event generated by an external actor.
- They are associated with system operations.
- A Controller is a non-user interface object responsible for handling a system event.
- A controller defines the method for the system event

Solution: Assign the responsibility for handling a system event message to a class representing one of the following choices:

- Represents the overall "system" (facade controller);
- Represents the overall business or organization (facade controller);
- Represents something in the real-world that is active (for example, the role of a person) that might be involved in the task (role controller);
- Represents an artificial handler of all system events of a use-case, usually named (use-case controller)

Example: In the POS application the current system operations have been identified as: endSale(), enterItem(), makePayment(). Who should be the controller for the system events such as enterItem and endSale?

- By the Controller pattern, here are the choices:
  - represents the overall "system": POST
  - represents the overall business or organisation: Store
  - somebody/something actively involved: Cashier
  - an artificial handler: BuyItemsHandler

Discussion:

- Normally, a controller object delegates to other objects the work that needs to be done to fulfill the responsibilities that it has been assigned.
- A facade controller usually handles all the system events of a system.
- Facade controllers are only suitable when there are only a few system events.
- Use case handler controller handles the system events of one use case.
- There are as many use case handler controllers as there are use cases.
- A use case controller is an alternative to consider when placing the responsibilities in any of the other choices of controller, leads to designs with low cohesion or high coupling (usually because of bloated controllers).
- External interfacing objects (for example window objects) and the presentation layer should not have responsibility for fulfilling system events: they delegate to a controller.

Advantages

- Increased potential for reusable components:
  - it ensures that business or domain processes are handled by the layer of domain objects rather than by the interface layer.
  - The application is not bound to a particular interface.

- Reason about the state of the use case:
    - As all the system events belonging to a particular use case are assigned to a single class, it is easier to control the sequence of events that may be imposed by a use case (e.g. MakePayment cannot occur until EndSale has occurred).

4. Don't Talk to Strangers (Law of Demeter)

Weak Form:

- Inside of a method M of a class C, data can be accessed in and messages can be sent to only the following objects:
    - this and super
    - data members of class C
    - parameters of the method M
    - object created within M
        - by calling directly a constructor
        - by calling a method that creates the object
    - global variables

Strong Form:

- In addition to the Weak Form, you are not allowed to access directly inherited members

Advantages:

- Coupling Control
    - reduces data coupling
- Information hiding
    - prevents from retrieving subparts of an object
- Information restriction
    - restricts the use of methods that provide information
- Few Interfaces
    - restricts the classes that can be used in a method
- Explicit Interfaces
    - states explicitly which classes can be used in a method

II. Problem

Apply two GRASP patterns in your project. Motivate your choice.

III. Bibliography

[1] Craig Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd Ed, Addison Wesley, 2004.

http://www.cs.bgu.ac.il/~oosd051/uploads/stuff/ApplyingUMLandPatterns.pdf