UNIT 3

DESIGN PATTERNS

GRASP

1. GRASP

- **General Responsibility Assignment Software Patterns (or Principles), abbreviated GRASP,** consists of guidelines for assigning responsibility to classes and objects in object-oriented design.

- The different patterns and principles used in GRASP are: Information Expert, Creator, Controller, Low Coupling, High Cohesion, Polymorphism, Pure Fabrication, Indirection, Protected Variations. All these patterns answer some software problem, and in almost every case these problems are common to almost every software development project. 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.

- It has been said that "the critical design tool for software development is a mind well educated in design principles. It is not the UML or any other technology". Thus, GRASP is really a mental toolset, a learning aid to help in the design of object oriented software.

2. Creator

- 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.

- In general, a class B should be responsible for creating instances of class A if one, or preferably more, of the following apply:

1. Instances of B contains or compositely aggregates instances of A
2. Instances of B record instances of A

3. Instances of B closely use instances of A

4. Instances of B have the initializing information for instances of A and pass it on creation.

3. Information Expert

- Information Expert is a principle used to determine where to delegate responsibilities. These responsibilities include 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. Information Expert will lead to placing the responsibility on the class with the most information required to fulfill it.

4. Low Coupling

- Low Coupling is an evaluative pattern, which dictates how to assign responsibilities to support:

1. low dependency between classes;
2. low impact in a class of changes in other classes;
3. high reuse potential

5. Controller

- The Controller pattern assigns the responsibility of dealing with system events to a non-UI class that represent 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. 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 use cases Create User and Delete User, one can have one UserController, instead of two separate use case controllers).

- It is defined as the first object beyond the UI layer that receives and coordinates ("controls") a system operation. The controller should delegate to other objects the work that needs to be done; 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 (assuming that the application has made an explicit distinction between the App/Service layer and the Domain layer) in an object-oriented system with common layers.

6. 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 element are strongly related and highly focused. Breaking programs into classes and subsystems is an example of activities that increase the cohesive properties of a system. Alternatively, low cohesion is a situation in which a given element has too many unrelated responsibilities.

- Elements with low cohesion often suffer from being hard to comprehend, hard to reuse, hard to maintain and adverse to change.

Polymorphism

According to Polymorphism, responsibility of defining the variation of behaviors based on type is assigned to the types for which this variation happens. This is achieved using polymorphic operations.

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 "Service" in Domain-driven design.

Indirection

- The Indirection pattern supports low coupling (and reuse 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.

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.

7. Visibility

Creational Design Patterns

⎫ In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design.

⎫ Creational design patterns solve this problem by somehow controlling this object creation. In object-oriented design, there is a notation of visibility for attributes and operations. UML identifies four types of visibility: public, protected, private, and package.

The UML specification does not require attributes and operations visibility to be displayed on the class diagram, but it does require that it be defined for each attribute or operation.

⎫ To display visibility on the class diagram, you place the visibility mark in front of the attribute's or operation's name. Though UML specifies four visibility types, an actual programming language may add additional visibilities, or it may not support the UML-defined visibilityies

Mark type	**Visibility**
+	Public
#	Protected
-	Private
~	Package

A BankAccount class that shows the visibility of its attributes and operations

⎫ Visibility - the ability of one object to “see” or have a reference to another object.

Visibility is required for one object to message another.

8. Applying GOF Design Patterns:

⎫ Design patterns represent common software problems and the solutions to those problems in a formal

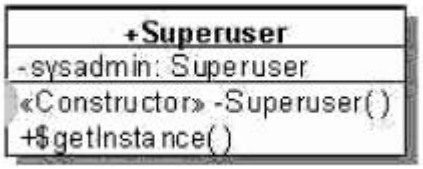
manner. They were inspired by a book written by architect Christopher Alexander. Patterns were introduced in the software world by another book: "Design Patterns: Elements of Reusable Object-Oriented Software", by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. These people were nicknamed the "Gang of Four" for some mysterious reason. The Gang of Four describes 23 design patterns.

With patterns you don't have to reinvent the wheel and get proven solutions for frequently encountered problems. Many books and articles have been written on this subject. This means that design patterns are becoming common knowledge, which leads to better communication. To summarize design patterns save time, energy while making your life easier.

8.1. Singleton

The singleton pattern deals with situations where only one instance of a class must be created. Take the case of a system administrator or superuser. This person has the right to do everything in a computer system. In addition we will also have classes representing normal users. Therefore we must ensure that these classes have no access to the super
user constructor. The solution to this problem in C++ and Java is to declare the superuser constructor private.

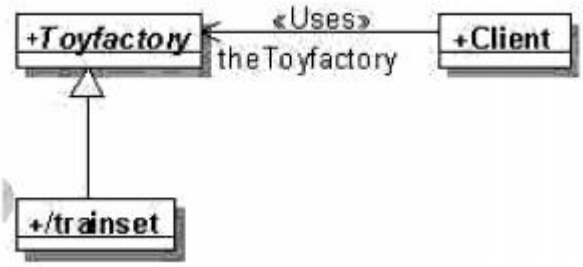
The super user class itself has a private static attribute sysadmin, which is initialised using the class constructor. Now we get an instance of the super user class with a public static method that returns sysadmin. Here is the class diagram:



8.2. Factory Method

The Factory Method pattern deals with situations where at runtime one of several similar classes must be created. Visualise this as a factory that produces objects. In a toy factory for instance we have the abstract concept of toy.

Every time we get a request for a new toy a decision must be made - what kind of a toy to manufacture. Similarly to the Singleton pattern the Factory Method pattern utilises a public static accessor method. In our example the abstract Toyfactory class will have a getInstance() method, which is inherited by its non abstract subclasses.

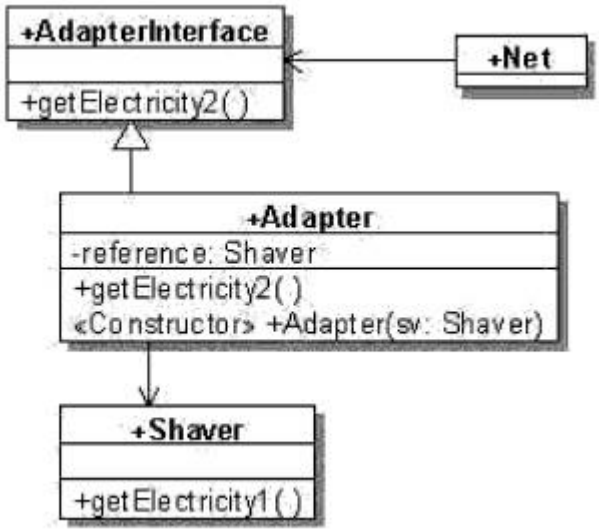


8.3. Adapter

- Sometimes it will have two classes that can in principle work well together, but they can't interface with each other for some reason. This kind of problem occurs when travelling abroad and you carry an electric shaver with you.

- Although it will work perfectly when you are at home. There can be problems in a foreign country, because of a different standard voltage. The solution is to use an adapter. Let's turn our attention back to the software domain. Here we will have an interface which defines new methods for example getElectricity2.

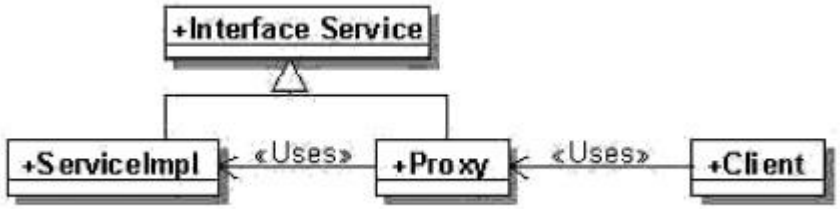
- An adapter class will wrap around the Shaver class. The adapter class will implement the interface with the new method.



8.4. Proxy

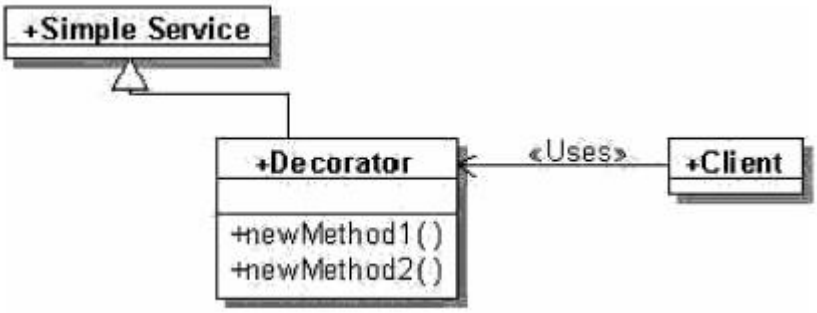
- The Proxy pattern deals with situations where you have a complex object or it takes a long time to create the object. The solution to this problem is to replace the complex object with a simple 'stub' object that has the same interface.

⟩ The stub acts as a body double. This is the strategy used by the Java Remote Method Invocation API. The reason to use the proxy pattern in this case is that the object is on a remote server causing network overhead. Other reasons to use the proxy can be restricted access (Java applets for example) or time consuming calculations.



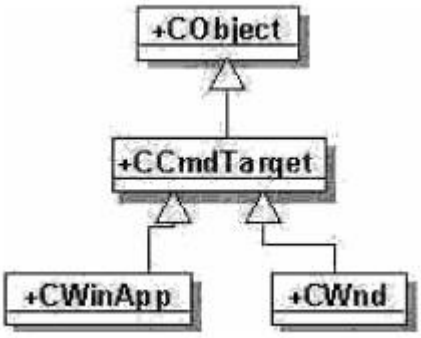
8.5. Decorator

⟩ The Decorator is usually a subclass, that is a body double for its superclass or another class with identical interface. The goal of the Decorator pattern is to add or improve the capabilities of the super class.



8.6. Composite

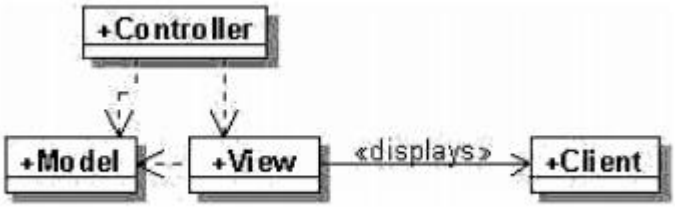
⟩ The composite is often encountered in GUI packages like for instance the Java Abstract Windwing Toolkit (AWT) or Microsoft Foundation (MFC) classes. All objects in this pattern have a common abstract superclass that descibes basic object conduct. The base class in the MFC hierarchy is CObject. It provides functions for debugging and serialization. All the MFC classes even the most basic ones inherit these facilities.



8.7. Observer and MVC

⎫ An application with Model - View - Controller setup usually uses the Observer Pattern. In a Java webserver environment the model will be represented by Java classes encompassing the business logic, the view is represented by Java Server Pages which display HTML in the client's browser and we will have a Servlets as Controllers.

⎫ The observer pattern strategy is to have view components take subscriptions for their model. This ensures that they will get notified if the model changes.

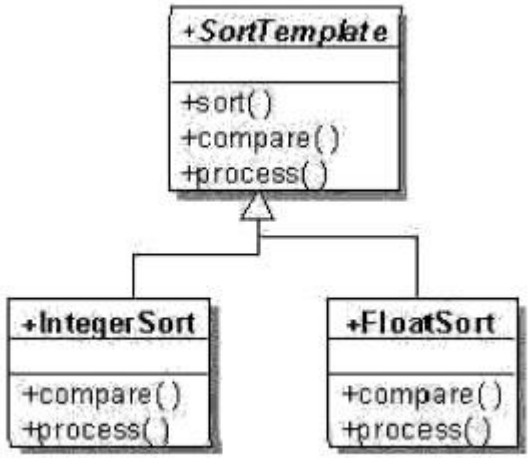


8.8. Template

⎫ In the good old days before OOP writing functions was the recommended thing to do. A sort algorithm would be implement by half dozen of functions, one sort function for integers, one sort function for floating points, one sort function for doubles etc.

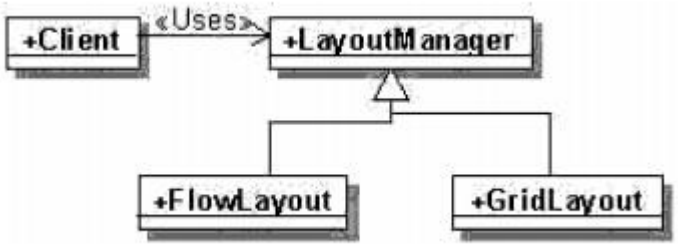
⎫ These functions are so similar that nobody in their right mind will type them letter by letter. Instead a programmer will write a template and copy the template several times. After that it's just a matter of writing down datatypes as appropriate. Thanks to OOP and the Template Design Pattern less code is required for this task.

⎫ Define an abstract Template class let's call it SortTemplate and it will have methods sort, compare and process (performs one cycle of the algorithm). Then we define classes for each datatype. These classes are subclasses of SortTemplate and implement the compare and process methods.



8.9. Strategy

⁾ The Strategy Design Pattern makes it possible chooses an implementation of an algorithm at run time. The implementation classes implement the same interface. In the case of the Java AWT Layout classes, the common interface is Layout Manager.



10. Summary

⁾ Design patterns are a hot research item. New patterns are emerging every day. In the future design patterns will be integrated in development tools. The main advantages of design patterns:

1. Provide proven solutions

2. Simplify complex problems

3. Improve communication

Classification and list

- Design patterns were originally grouped into the categories: creational patterns, structural patterns, and behavioral patterns, and described using the concepts of delegation, aggregation, and consultation.
- For further background on object-oriented design, see coupling and cohesion, inheritance, interface, and polymorphism. Another classification has also introduced the notion of architectural design pattern that may be applied at the architecture level of the software such as the Model-View-Controller pattern.

Creational Design Patterns

- In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

- The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation