Design patterns description

1) Singleton pattern (creational pattern):

Context: It is very common to find classes for which only one instance should exist (singleton).

Problem: How do you ensure that it is never possible to create more than one instance of a user class. And provide a global point of access to it.

Forces: The use of a public constructor cannot guarantee that no more than one instance will be created. user instance must also be accessible to all classes that require it; therefore it must often be public.

Solution: Have the constructor private to ensure that no other class will be able to create an instance of the class user. Define a public static method called “Get user access ()”, The first time this method is called , it creates the single instance of the class “user” and stores a reference to that object in a static private variable called “the user

Example : Applying on user Class

If (the User==null)

the User=new User;

return the User;

|  |
| --- |
| User |
| theUesr:user |
| -user ():void <<private>>  +GetUseInstance():Amin |

2) Delegation pattern (structrual): “applying on many classes in the project”

Context: You are designing a method in a class. You realize that another class has a method which provides the required service. Inheritance is not appropriate (e.g., because the is-a rule does not apply).

Problem: How can you most effectively make use of a method that already exists in the other class?

Forces: You want to minimize development cost by reusing methods.

Solution: The delegating method in the delegator class in our example (sebsor,sensor reading , power supply classes ) calls a method in the delegate class in our example (status class)to perform the required task. An association must exist between the delegator and delegate classes

Example:

Delegate class Delegator classes

|  |
| --- |
| Status |
|  |
| setstatus (string):boolean |

|  |
| --- |
| Counselor |
|  |
| sendstatus (string):boolean |

|  |
| --- |
| Student |
|  |
| sendstatus(string):boolean |

3) OBSERVER

Context: ▪When partitioning a system into individual classes you want the coupling between then to be loose so you have the flexibility to vary them independently.

Problem: ▪A mechanism is needed to ensure that when the state of an object changes related objects are updated to keep them in step.

Forces: ▪The different parts of a system must kept in step with one another without being too tightly coupled

Solution: ▪ One object has the role of the subject/publisher and one or more other objects the role of observers/subscribers.

The observers register themselves with the subject, & if the state of the subject changes the observers are notified & can the update themselves.

▪ The are two variants: ▪the Push Model where the subject send the observers detailed information about the change that has occurred, and

▪the Pull Model where the subject simple notifies the observers that there have been changes, and it's the responsibility of the observers to find out the details they need to update themselves

Example:

Observable observer

|  |
| --- |
| sensor |
|  |
| setstatus (string):boolean |

|  |
| --- |
| Alarm |
|  |
|  |

|  |
| --- |
| Movement sensor |
|  |
|  |

|  |
| --- |
| voltage sensor |
|  |
|  |

|  |
| --- |
| Door sensor |
|  |
|  |