DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
CS6502-OBJECT ORIENTED ANALYSIS AND DESIGN

UNIT-II
DESIGN PATTERNS

GRASP: Designing objects with responsibilities – Creator – Information expert – Low Coupling – High Cohesion – Controller - Design Patterns – creational - factory method - structural – Bridge – Adapter - behavioral – Strategy – observer.

PART- A

1. How to Choosing the Initial Domain Object?

Choose as an initial domain object a class at or near the root of the containment or aggregation hierarchy of domain objects. This may be a facade controller, such as *Register,* or some other object considered to contain all or most other objects, such as a *Store*

.

2. How to Connecting the UI Layer to the Domain Layer?

• An initializing routine (for example, a Java *main* method) creates both a UI and a domain object, and passes the domain object to the UI.

• A UI object retrieves the domain object from a well-known source, such as a factory object that is responsible for creating domain objects.

3. Mention the Interface and Domain Layer Responsibilities.

The UI layer should not have any domain logic responsibilities. It should only be responsible for user interface tasks, such as updating widgets. The UI layer should forward requests for all domain-oriented tasks on to the domain layer, which is responsible for handling them.

5. Define patterns.

A pattern is a named problem/solution pair that can be applied in new context, with advice on how to apply it in novel situations and discussion of its trade-offs.

6. How to Apply the GRASP Patterns?

The following sections present the first five GRASP patterns:

. Information Expert
. Creator
. High Cohesion
. Low Coupling
. Controller

7. Define Responsibilities and Methods.

The UML defines a responsibility as "a contract or obligation of a classifier" [OMG01]. Responsibilities are related to the obligations of an object in terms of its behavior. Basically, these responsibilities are of the following two types:

- knowing
-doing

Doing responsibilities of an object include:

- doing something itself, such as creating an object or doing a calculation
- initiating action in other objects
-controlling and coordinating activities in other objects
Knowing responsibilities of an object include:
- knowing about private encapsulated data
- knowing about related objects
- knowing about things it can derive or calculate

8. Who is creator?
Solution Assign class B the responsibility to create an instance of class A if one or more of the following is true:
. B *aggregates* an object.
. B *contains* an object.
. B *records* instances of objects.
. B *closely uses* objects.
. B *has the initializing data* that will be passed to A when it is created (thus B is an Expert with respect to creating A).
B is a *creator* of an object.
If more than one option applies, prefer a class B which *aggregates* or *contains* class A.

9. List out some scenarios that illustrate varying degrees of functional cohesion.
-Very low cohesion
-low cohesion
-High cohesion
-Moderate cohesion

10. Define Modular Design.
Coupling and cohesion are old principles in software design; designing with objects does not imply ignoring well-established fundamentals. Another of these. Which is strongly related to coupling and cohesion? is to promote modular design.

11. What are the advantages of Factory objects?
• Separate the responsibility of complex creation into cohesive helper objects.
• Hide potentially complex creation logic.
• Allow introduction of performance-enhancing memory management strategies, such as object caching or recycling.

12. Designing for Non-Functional or Quality Requirements.
Interestingly—and this a key point in software architecture—it is common that the large-scale themes, patterns, and structures of the software architecture are shaped by the designs to resolve the non-functional or quality requirements, rather than the basic business logic.

13. Abstract for Factory (GoF) for Families of Related Objects.
The Java POS implementations will be purchased from manufacturers.
For example5:
// IBM's drivers
com.ibm.pos.jpos.CashDrawer (implements jpos.CashDrawer)

com.ibm.pos.jpos.CoinDispenser (implements jpos.CoinDispenser)
// NCR's drivers
com.ncr.posdrivers.CashDrawer (implements jpos.CashDrawer)
com.ncr.posdrivers.CoinDispenser (implements
jpos.CoinDispenser)

14. What is meant by Abstract Class Abstract Factory?
A common variation on Abstract Factory is to create an abstract class factory that is accessed using the Singleton pattern, reads from a system property to decide which of its subclass factories to create, and then returns the appropriate subclass instance. This is used, for example, in the Java libraries with the *java.awt.Toolkit* class, which is an abstract class abstract factory for creating families of GUI widgets for different operating system and GUI subsystems.

15. What is meant by Fine-Grained Classes?
Consider the creation of the *Credit Card, Drivers License,* and *Check* software objects. Our first impulse might be to record the data they hold simply in their related payment classes, and eliminate such fine-grained classes. However, it is usually a more profitable strategy to use them; they often end up providing useful behavior and being reusable. For example, the *Credit Card* is a natural Expert on telling you its credit company type (Visa, MasterCard, and so on). This behavior will turn out to be necessary for our application.

16. Define coupling. APIRAL/MAY-2011
The degree to which components depend on one another. There are two types of coupling, "tight" and "loose". Loose coupling is desirable for good software engineering but tight coupling may be necessary for maximum performance. Coupling is increased when the data exchanged between components becomes larger or more complex.

17. What is meant by Low Coupling?
Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. An element with low (or weak) coupling is not dependent on too many other elements; "too many" is context-dependent, but will be examined. These elements include classes, subsystems, systems, and so on.

18. What is meant by High cohesion?
Cohesion (or more specifically, functional cohesion) is a measure of how strongly related and focused the responsibilities of an element are. An element with highly related responsibilities, and which does not do a tremendous amount of work, has high cohesion. These elements include classes, subsystems, and so on.

19. Define Controller.
Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:
- Represents the overall system, device, or subsystem *(facade controller).*
- Represents a use case scenario within which the system event occurs, often named <UseCaseName>Handler, <UseCaseName>Coordinator, or <Use-CaseName>Session *(use-case or session controller).*

- Use the same controller class for all system events in the same use case scenario.
-Informally, a session is an instance of a conversation with an actor.
-Sessions can be of any length, but are often organized in terms of use cases (use case sessions).

PART- B

1. Explain Grasp: designing objects with responsibilities.
-Responsibilities and Methods
-Responsibilities and Interaction Diagrams
-Patterns

2. Explain GRASP: Patterns of General Principles in Assigning Responsibilities. **APIRL/MAY-2011**

-The UML Class Diagram Notation
-Information Expert (or Expert)
-Creator
-low coupling
-high cohesion
-controller
-object design and CRC CARDS

3. How to Determining the Visibility of the Design Model?
-Visibility between Objects
-Visibility

4. Explain about Patterns for Assigning Responsibilities.
-Polymorphism
-Pure Fabrication
-Indirection
-Protected Variations

5. Designing the Use-Case Realizations with GoF Design Patterns. **APRIL/MAY-2011**
-Analysis" Discoveries during Design: Domain Model
-Factory
-Singleton
-Conclusion of the External Services with Varying Interfaces Problem 3
-Strategy
-Composite
-Facade