Paradigm - Rules, principles and concepts

Imperative programming we tell how to do

Object oriented programming

1. Encapsulation - Contract trade in NSE
2. Abstraction - Trading system
3. Inheritance - Future contract, Option contract all extends from Contract
4. Polymorphism - Collateral

Declarative programming - we till what do to without saying how to do it

Functional programming - sub type of declarative programming

Relationships

1. Inheritance (IS-A)
2. Generalization

When more than 1 class extends from the base class, the base class is called generalization of sub class

and each sub class is specialization of base class

1. Realization

When an interface is implemented by a class

1. Dependency (USES-A)

When a class is dependent on class but that class is not dependent, changes to dependant class will result change in other class (The dependant class is used as parameter in method)

1. Association
   1. Aggregation (HAS-A)

Department has Teacher, but if Department does not exist Teacher can exist

Dependant class is used as instance variable

1. Composition (PART-OF)

House has Rooms, but if House is destroyed Room also gets destroyed

**Template pattern**

Defines skeleton of algo in a method

Template pattern uses Inheritance

Forces subclasses to inherit template methods

Template class is an abstract class with template method as final method

Example Report generation code with different output formats

Template method is final method that defines the order of invocation of methods, method implementation is specific to sub-classes and declared abstract

Subclasses are different file formats (CSV,TXT,HTML)

Hooks - methods defined in template class with no or default implementation, they are used to control the algo

Method to check if transfer required

Arrays.sort( uses compareTo() method which is abstract)

JdbcTemplate in Spring

Pros

Code reusability

No duplication of code

Cons

If anything changes in template method, all subclasses needs to change

Possible scenarios when template methods are not enough to handle business case (JSON exporter)

Define interface with template method and template class extends this interface, so that subclass not wanting to use the template method can define its own way of invocation

**Singleton pattern**

Runtime class

Exceptions should be handled at the presentation layer

Dependency inversion is the alternative to singleton

Problems with singleton

1. Difficult to mock singleton class so difficult to test

1. Tight coupling

2. Violation of single responsibility principle, it is managing its lifecycle as well

3. Violation of Open Close principle - closed for changes

Making singleton objects thread safe

1. using static synchronized methods //performance impact

2. using eager initialization // cannot be used if object creation takes time

public static volatile Singleton instance=new Singleton()

3. using double check locking with volatile and synchronized

public static volatile Singleton instance=null;

if(instance==null){

synchronized(Singleton.class){

if(instance==null){

instance=new Singleton();

}

}

}

**Strategy pattern**

Uses composition

Similarities with template pattern

Use of abstraction

Algorithm/strategy can selected at runtime

Differences

Code reusability (steps are reused in template, common methods are put in template class), strategy no reusability

**Factory pattern**

**Simple**

private constructor

final class

static factory methods

Don't handle exceptions in factory method always propagate

ex: Executors

**Factory method**

Defer object creation to subclasses, define factory method as abstract, client will get to see abstract not actual implementing class

abstract method is not necessary only required is object creation should be deferred to sub class

Uses inheritance

**Abstract Factory**

Uses composition

The factory method is moved to another class with diff subclasses providing different implementation for factory method

then use this abstract class with composition

Dependency inversion

Combine related factory classes to created single factory which provides multiple related objects

**Facade pattern**

Facade is entry point to other classes

Clients classes use facade to access other classes

Reduces coupling

Enables encapsulation

Should not restrict the client from accessing systems directly(in case of complex operations this may be restricted)

Facade should not provide any functionality that is not provided by sub systems

Client--Facade--Sub systems

**Observer Pattern**

Publisher manages the state and in case of state change updates the Subscribers

Java's built in support

Observable - Subject

Observer

The subject need not know about the implementations of observers, and code in suject need not be changed for new observers added

Observable provides both push and pull mechanism

setChanged() needs to be called before notifying observers

notifyObservers() - pull         update(Observable o,Object arg) arg is null in case of pull

notifyObservers(arg) - push

**Decorator pattern**

Decorator class extends the same class the object it decorates extends

InputStream - Component

FileInputStream - concrete implementation of InputStream

FilterInputStream - abstract decorator (needed if all decorators have some common code or for enforcing functionality to decorators)

BufferedInputStream - Decorator

Large number of smaller classes

**State pattern**

It provides different operations the context can do based on the state

Each state class has reference to the context object

The context object defines the different operations that can be done based on the state

**Adapter pattern**

3 pin socket is the Adaptee

2 pin plug is the target interface

Adapter will implement the target interface and have instance of Adaptee interface

Types of Adapters

1. Object Adapters (default)

The adapter class has a reference of Adaptee (composition)

1. Class Adapters (requires multiple inheritance)

The adapter class extends both Adaptee(inheritance) and Target

In legacy application use of Enumerators in collections such as Vector for iteration

However new collection like ArrayList support Iterator, if legacy code iterates collection enumerator, to iterate through a ArrayList we need to have adapter which implements Enumerator(Target) and uses Iterator(Adaptee)

**Façade pattern**

The difference between Adapter and Façade is their intent

Intent of adapter is to convert interface of class to an interface that client expects but façade provides simple interface to subsystem, client can directly access the subsystem

Both decouple client from the subsystem

**Command pattern**

Encapsulate request as object to decouple the client and receiver

Client creates a command object which contains the reference of Receiver, it defines an execute() methods in which receiver methods are called

The client then sets the command object on a invoke by calling the setCommand method.

The invoker may/may not then call the commands execute method

Example of command pattern is submitting tasks to thread

Thread -> Invoker

Client -> Code creating instance of Runnable or Callable

Runnable/Callable -> Command interface

Task that implements Runnable/Callable - > Command object (no receiver as runnable itself is doing that)

Meta command pattern - Invoking multiple receiver methods with single command object

**Proxy Pattern**

Creates another object of the actual object thereby controlling access to actual object

**Types**

Remote proxy (RMI) - provide access to remote object

Virtual proxy - used for creating of resource heavy object

Protection proxy - used to control access to object based on access rights (Runtime proxy)

Both the proxy and actual object implement the same interface

In case of remote proxy call to proxy object is translated as call to remote object

In case of virtual proxy, remote proxy method is used to create instance of actual object and after the object is created calls to proxy are delegated to actual object

**Protection proxy**

Subject - the object we want to control access to

Proxy - created at runtime from Proxy.newProxyInstance() which accepts subject's classloaders, interfaces and an Invocation handler

Invocation handler - On calling methods on the proxy it delegates the call to invoke() on the invocation handler

which in turns calls the method on the subject based on some condition , it has reference to the subject

**Iterator pattern**

Used to access elements of collection without exposing underlying implementation

The logic for iterator is specified in the iterator implementation rather than the collection (Single responsiblity)