Singleton Design pattern

Intent :- " a class has only one instance while providing a global access point to the instance "

what do you mean by that?

(1) → There will be only one object of class in entire application (cannot have multiple object)

(2) → Instead of creating a new object, using the old one which already have been created

(3) → global access point means the instance can be accessed from anywhere in the application

2. Motivation

what is the motivation of singleton design pattern?

→ motivated by the need to ensure a that only one instance of a class exists in a program providing a global access point to that instance.

→ To control the instantiation process of class & limit the number of instance of class to one

→ Common Reason
- → Resource sharing
- → Global access
- → controling object creation

3 **Applicability :-**

what is the applicabity of singleton, where to use it ?

(i) Resource management :- when you have a limited resource that need to be shared in molliple parts of system.

Ex:- Database Connection object, File system access, a thread pool

(ii) Global access :- If a class need globally accessible throught the application

Ex:- a logger, a cache, configuration manager

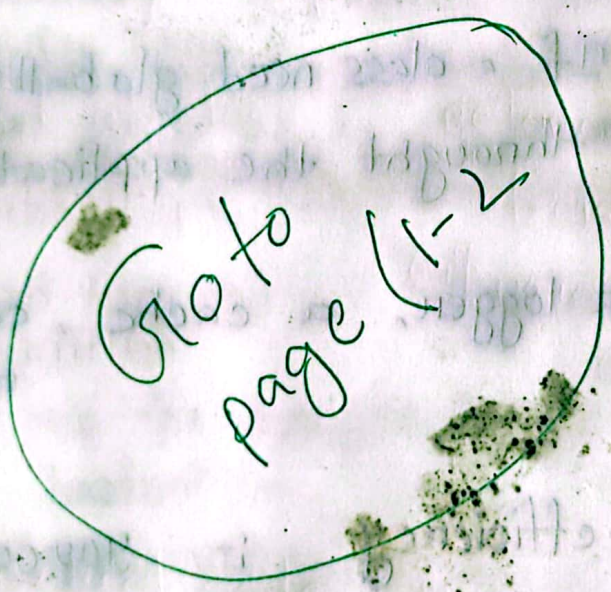
(iii) less costly and efficiency System :- you can instantiate the object once and reuse it throught the program

Pros (Problem that singleton solves)

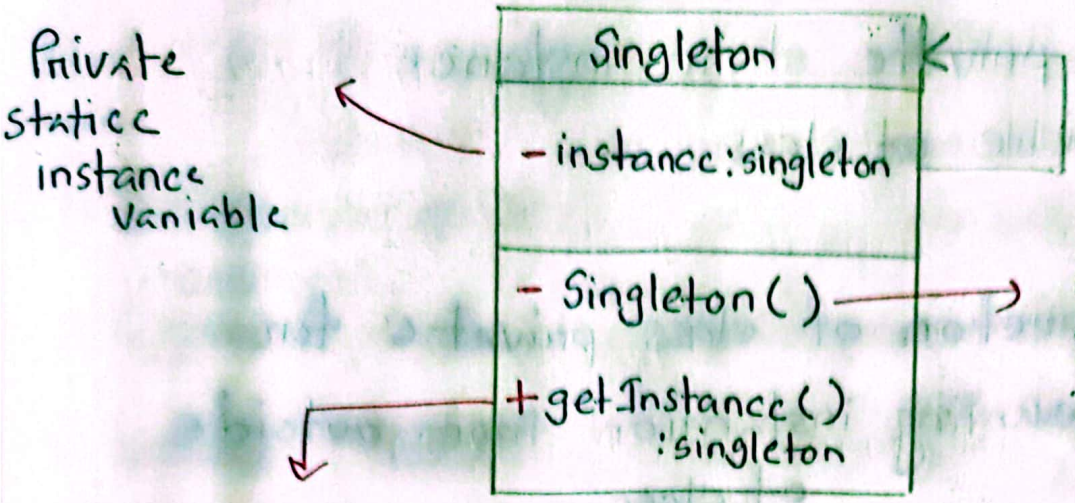
1. ensure a class has just single instance
2. provide a global access point to the instance
3. object is initialize only once

cons

1. Violates Single Responsibility principle



UML Diagram of singleton



declares a public static getinstance() method that returns the same instance of its own class (only way of getting singleton object)

Q:- What are participant object of singleton ?

1. Singleton :- provides getinstance()
2. Client :- - request to the instance of singleton using getinstance()
   - use it to access or perform operation resource

How to implement singleton design pattern?

1. Create a private static instance variable of class

2. Make construcion of class private for preventing instantion from outside of class

3. providing a public static getinstance() method of class ; returning the instance of class
(if no instance, create a new instance first)

4. In client code, now call getinstance() method of class instead of direct calls to constructor

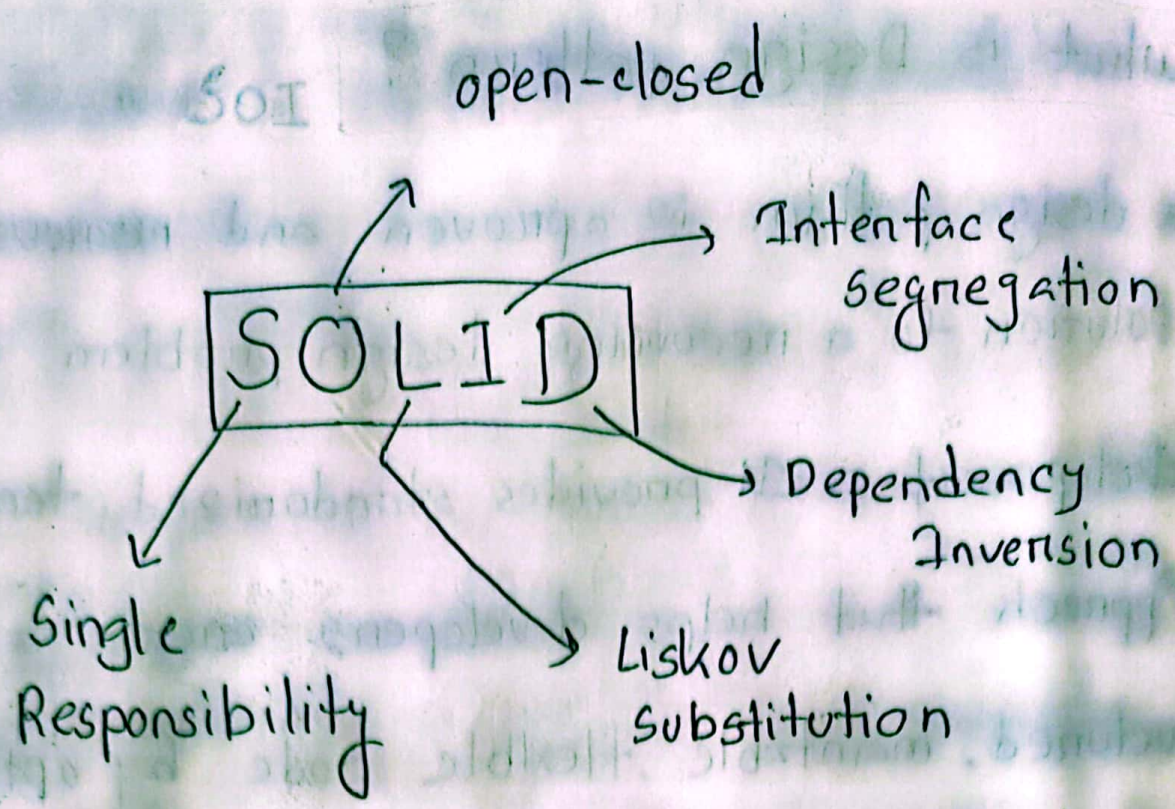
what is Design pattern ?

a design pattern is a proven and reusable solution to a recurring design problem in softwa development . It provides standarized template or approch that helps developens create a well structured, maintable, flexible code by optimizing the best pratices and collective experience

what does pattern consist of ?

1. Intent (describes pattern with problem and solution)
2. Motivation
3. Structure (uml Diagram)
4. code Example

Principles


open-closed
SOLID
Intenface segregation
Dependency Inversion
Single Responsibility
Liskov substitution

(SRP) Single Responsibility Principle

Principle :- states that a class should have only one reason to change

multiple Responsibility

what do you mean?

a class have only single responsib- on single functionality on single purpose of job

```
publicc class BankServicc
{
  public void login (String User, int pass)
  {
  }
  public void deposit (string Account, int amount)
  {
  }
  public void withdraw (string Account, int amount
  {
  }
}
```

Here, BankService class has the responsibility of login, deposit and withdraw. It means it has more that one responsibility which violates the single Responsibility principle.

To achieve the goal of SRP Here,
You need to divide the responsibility to different class

∴ Bank class divided into 3 different class

1. BankService Login class
2. BankService Deposit class
3. BankService Withdraw class

New code

① Class-1

```
public class BankService_Login {
    public void login (string user, int pass) { }
}
```

② Class-2

```
public class BankService_Deposit {
    public void deposit ( string Account, int amount)
    { }
}
```

③ Class-3

```
public class BankService_withdraw {
    public void withdraw ( string Account, int
                                        amount)
    { }
}
```

(ocp) open closed principle

principle states that software entities (class, modules, functions) should be open for extensions but closed for modification

→ what mean?

able ta extend class behaviour without modifying it

need to modify if want any extentension

```
Public class payment {

    public void bkash ( string id , int amount, int pass)
    {
    }

    public void card ( string id , int amount, int pss)
    {
    }
}
```

Here, if i want to add "nagad payment" in payment service (to add extension), I need to modify the payment service class. which violates open closed principle.

To achieve the goal of open-closed principle :-

① Creating a payment service Interface which can be easily extended without modifying in future

```
public Interface paymentservice {

    public void payment (id, pass, amount)
    {
    }
}
```

② Creating New payment classes and extending payment service without modifying it.

```
public class bkash implements paymentservice {

    public void payment ( id, amount, pass )
        { // proced bkash service
        }
    }

public class cand implements paymentservice
    {
    public void payment ( id, amount, pass )
        { // proceed cand
        }

public class nagad implements paymentservice
    {
    public void payment ( id, amount, pass )
        {}
    }
```

LSP (Liskov Substitution Principle)

Principle states " The object of superclass should be replaceable with the objects of it subclass without breaking the application ".

Derived / child classes must be substitutable for base/parent class

ensure inheritance used properply

What Do you mean?

object of subclass should behave the same way.. as the object of superclass

NOTE → If an overnide method does nothing on just throw exception then it probably violating Liskov.

Parent class → Public class Bird {

Bird has two methods

```
public void flying()
{ // flying }

public void walking()
{ // walking // }

}
```

Subclass →

```
public class Dove extends Bird{
{
    public void flying()
    { // Dove fly }

    public void walking()
    { // Dove walk }

}
```

Dove is a child class of Bird

As dove can both fly and walk. Dove is a complete substitute of Bird class

∴ doesn't break LSP

Subclass

penguine subclass of bind. But penguine can't fly so it is not complete substitution of Bind class

Breaks LSP

```
public class penguine
    extends Bind {

public void flying()
    { throws exception }

public void penguine
    walking()
    { // penguine walk }

}
```

Soluting the whole code

①

```
public class Bind {
public void walking( )
    { // walking }
}
```

②

```
public class flyingBind
    extend Bind {
public void flying( )
{ //flying }
}
```

③

```
public class
Dove extends flyingBind {
    // can fly and walk methods
}
```

④

```
public class penguine extends Bind {
    // can walk , no fly
}
```

LSP

① ensures subtypes can be placed in their base type without breaking the application

② focus on inheritance is used correctly

③ concerned with inheritance & polymorphism

ISP

① ensure that interface are designed in a specific and relevant way to the needs of client

② focus on avoiding unnecessary interface implementation

③ concerned with interface and their usages

ISP (Interface segregation principle)

principle states that client should not be forceed to implement methods of an interface, they don't use

Social media interface

```
public interface socialmedia {

    public void chatwithfriend();

    public void post();

    public void sendphoto();

}
```

Facebook (as client) using the functionalites of social media

```
public class Facebook implements SocialMedia {

    public void chatwithfriend()
        { // chat FB // }

    public void post()
        { // posting // }

    public void sendphoto()
        { // sending // }
}
```

Here Facebook as client has all feature of this.

Not violating ISP

whatsapp using the functionalities

```
public class whatsapp implements SocialMedia {

    public void chatwithFriend()
    {
    }

    public void post()
    {
        cannot post
        in whatsapp
    }

    public void sendphoto()
    {
    }
}
```

As whatsapp don't have posting feature. Interface is forcing whatsapp to implement post() method

Violating ISP

Solution

① public interface SocialMedia → now it contains methods only both facebook and whatsapp can use

```
{
    public void chatwithfriend();
    public void sendphoto();
}
```

② public interface PostManagen → new interface for post method

```
{
    public void post();
}
```

③ public class facebook implements SocialMedia, PostManagen → facebook use both interfaces

```
{
}
```

④ public class whatsapp implements SocialMedia → whatsapp only use socialmedia cause it can't post

```
{
}
```

DIP

Dependency Inversion principle

Go to page-3

Go to page-4

Factory Design pattern

Intent :- " Provide an interface for creating objects in a superclass but allow the subclass to alter the type of object that will be created "

↓ what do you mean?

→ define an interface or abstract class for creating object

→ Let the subclass decide which class to instantiate

Motivation the motivation of factory design pattern?

→ motivated by the need of creating object without specifying the exact class of object that will be instantiated.

→ The goal is to encapsulate object creation and provide a level of abstraction.

1. Decoupling object creation
2. Abstraction and encapsulation
3. Simplifying object creation logic
4. Flexibility

Applicability

when and where to use factory method?

① Use factory method when dont know since before the exact type and dependencies of the object your code should work with.

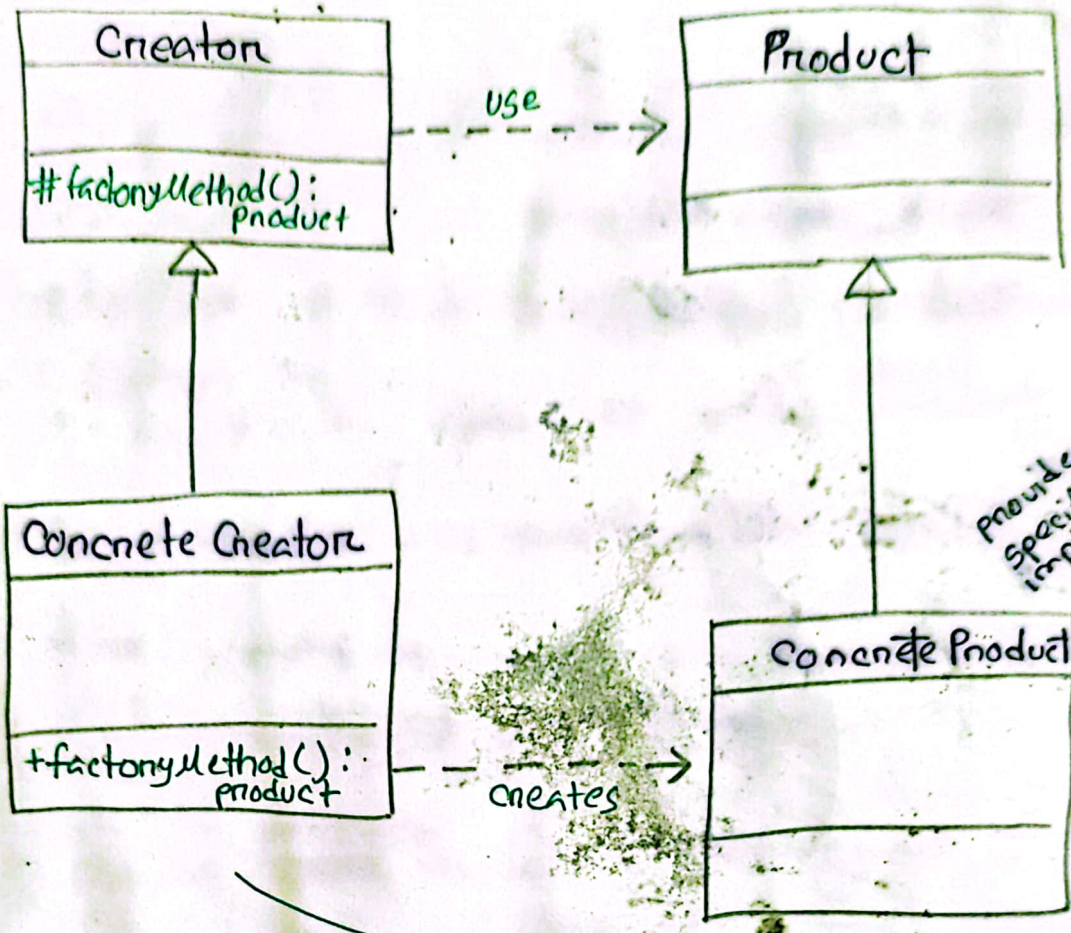
(factory method separates product construction code that actually uses the product)

② Use when you want to provide users of your library / framework with a way to extend its internal components

③ Use factory method when you want to save system resources by reusing existing object instead of building them each time

declare factoryMethod()
for creating products

(common interface
for the objects
created by factory)

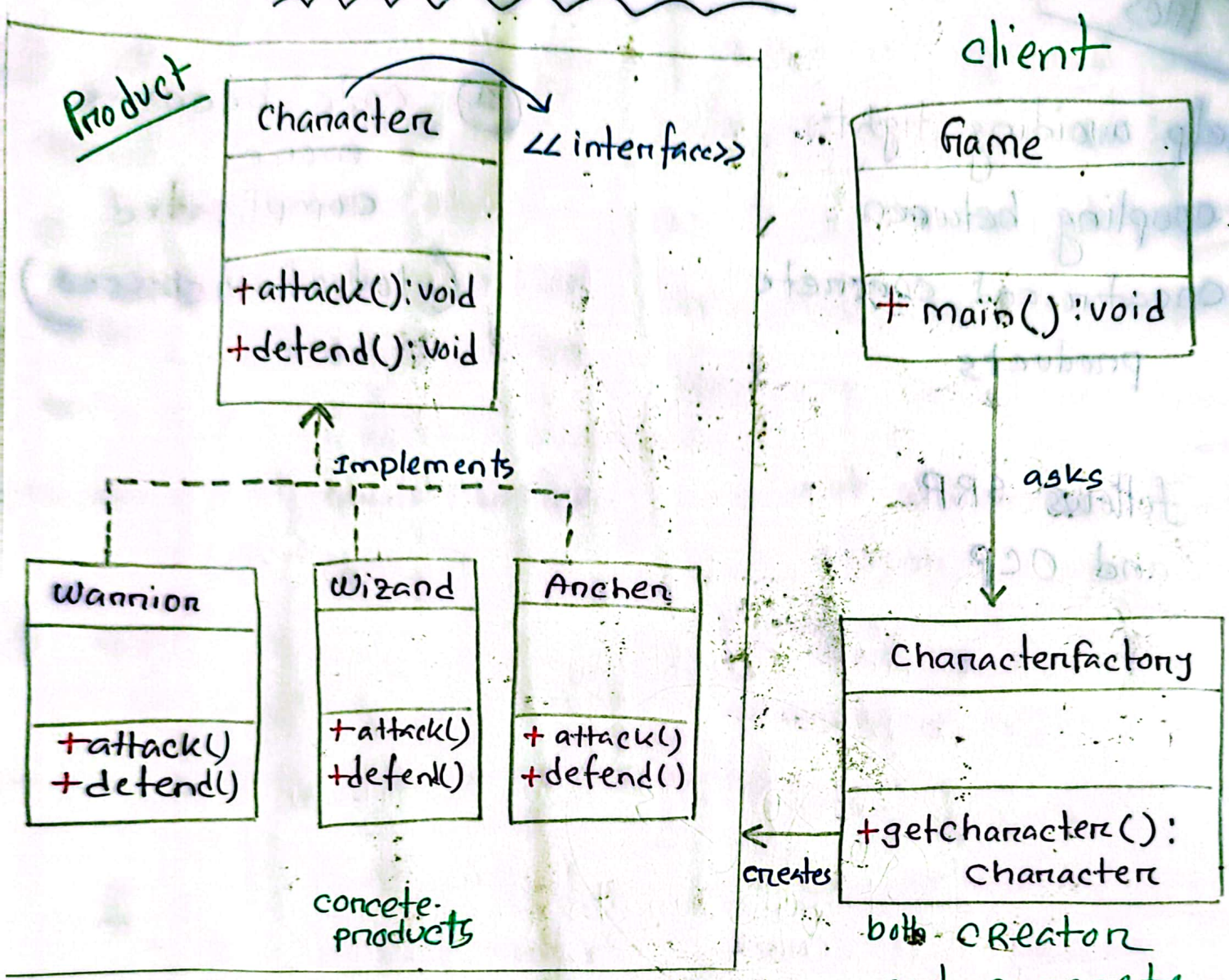


(override the
factory methods
to create specific
concrete product
objects)

Bestron™

③ character — Warrior
— Wizard
— Archer

UML case Diagram



what are the participant object of factory design pattern?

1. Game
2. characterfactory
3. character — warrior, wizard, Archer

Factory Design

1. It aims to encapsulate object creation by providing separate factory class responsible for creating an object of particular type

2. It provides a way to create object without specifying their concrete class

3. consists of a factory interface or abstract class that declare a factory method for creating object

Abstract factory

1. It aims to provide to provide an interface for creating families of related or dependent object without specifying concrete classes

2. It abstracts the creation of multiple related objects ensuring created objects are compatible with each other

3. consists of an abstract interface or class that declare factory method for creating families of Related object

Pros

1. help avoiding tight coupling between creator and concrete products

2. follows SRP and OCP

Cons

1. code become more complicated (lot of sub classes)

Go to page-5