

Patterns:

1. strategy
2. Decorator
3. Observer
4. Template method
5. Prototype
6. (spare exam 2010) + Fahim's note
7. Iterator
8. Adapter

Handwritten Ans/SWERE for Semester Final

Design Patterns

The Strategy Pattern: Strategy Pattern is a behavioral pattern that defines a family of algorithms, encapsulate each one and allows to select an algorithm at runtime.

Ques: Why is it called Strategy pattern?

Ans: Because, at runtime, we can select an algorithm/strategy to solve a particular problem.

Q: কেন / কোন situation এ strategy প্রিয়?

Ans:

i) If object এর behavior যখন runtime কে change/switch করা লাগবে তেন specific task perform করার জন্ম।

ii) যখন অনেকগুলো similar class থাকবে এবং প্রিটি class একটি particular behavior execute করবে তখন \Rightarrow duplicate code এর সুষ্ঠি হবে।

strategy pattern একটি class hierarchy
maintain বজায় রাখলে duplicate code
reduce করতে সাহায্য করে। So, duplicacy problem
solve করে।

(iii) Runtime এ easily ~~state~~ strategy switch

to inherit করায় অস্তু।

two or more class inheritance, multiple inheritance

2. How do we implement this?

i) Maintains idea হল common behaviors প্রেরণা
অবস্থা inherit করতে পারবে আবেগকে একটি

class এ রাখবে। এবং uncommon দেখ

behaviours এর upon depend করে প্রতিটোর

অস্তু interface create করবে।

(ii) প্রতিটো আমরা behaviour গুলোকে encapsulate

করবে এবং মেঝেন class, interface গুলোকে

implement করবে। একটো class গুলোকে

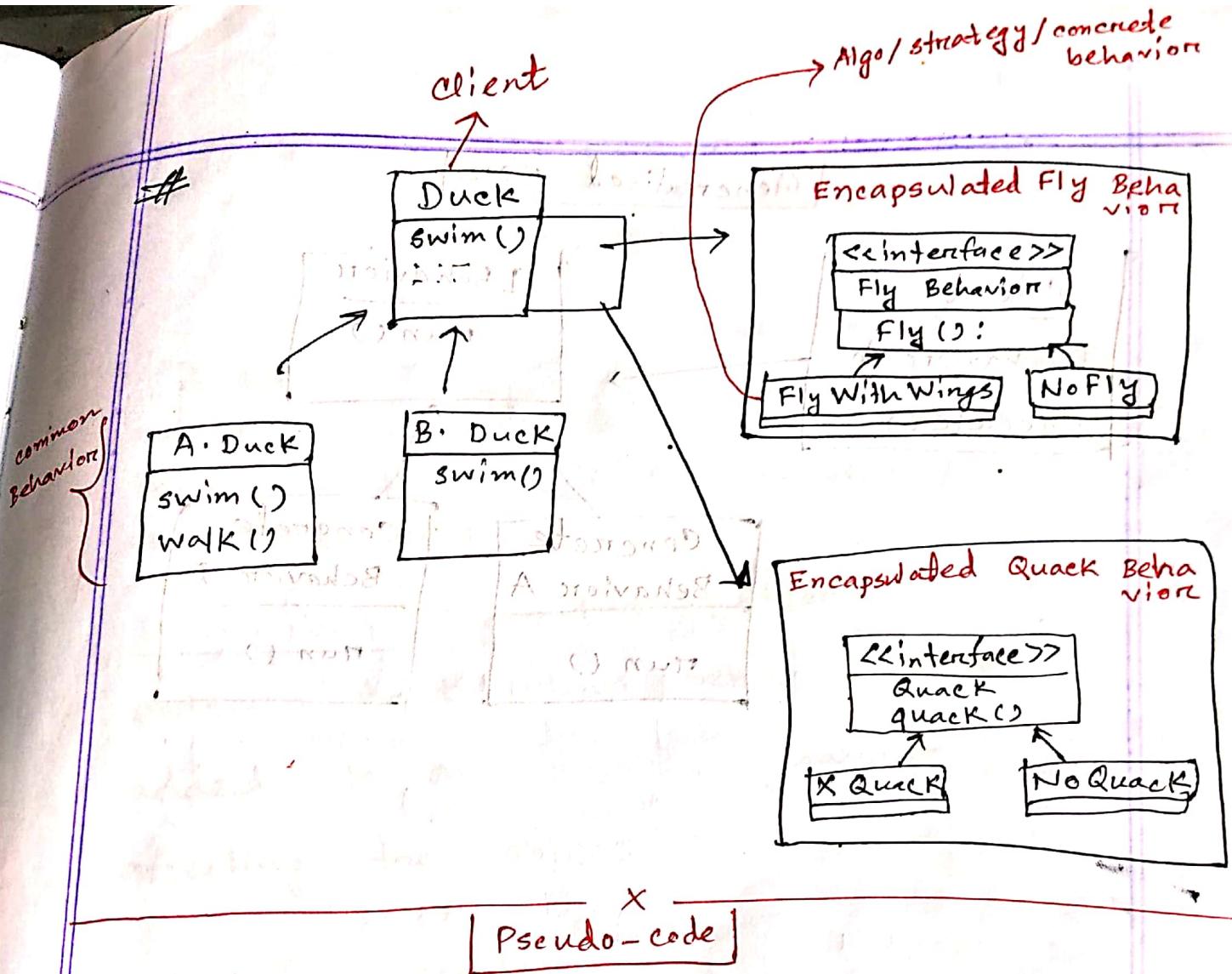
concrete behaviors কিন্তু বলে।

বিষয় ক্ষেত্র কাজ করবে।

একই প্রকার কাজ করবে।

বিষয় ক্ষেত্র কাজ করবে।

বিষয় ক্ষেত্র কাজ করবে।



Pseudo-code

Class Duck {

~~I~~ I Fly Behav fb;

I Quack Behav qb;

public Duck (IFlyBehav fb, IQuackBehav qb) {

this. fb = fb;

this. qb = qb;

}

}

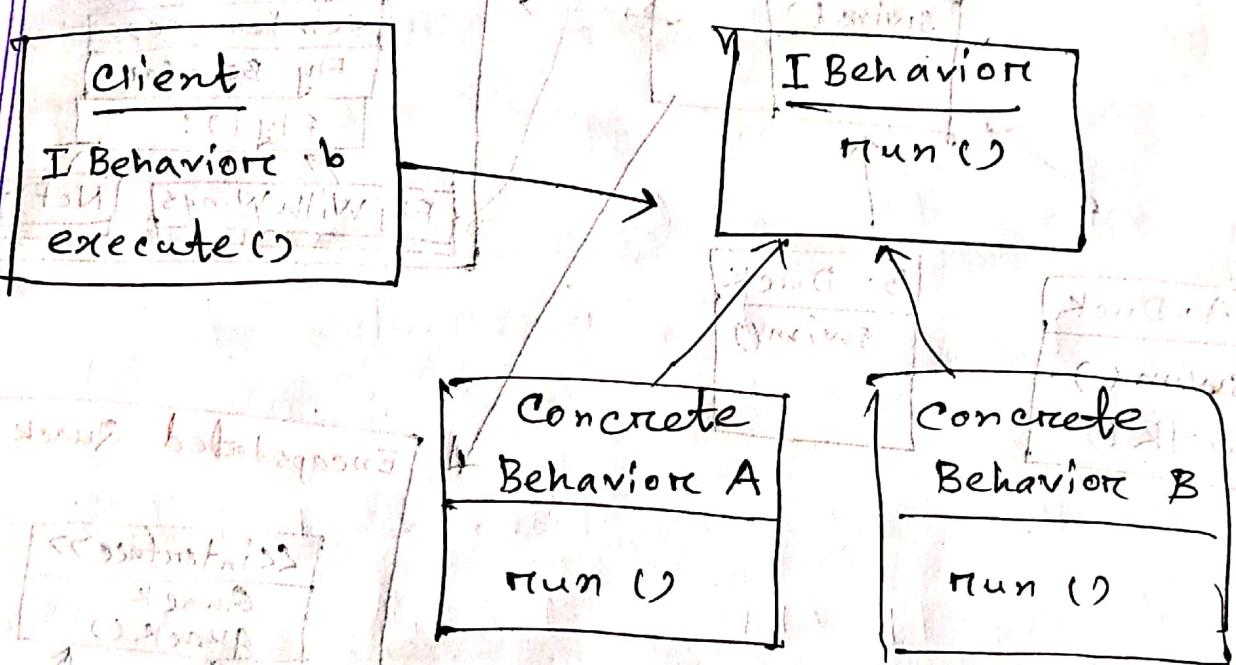
Public void Fly () {

this. fb. fly();

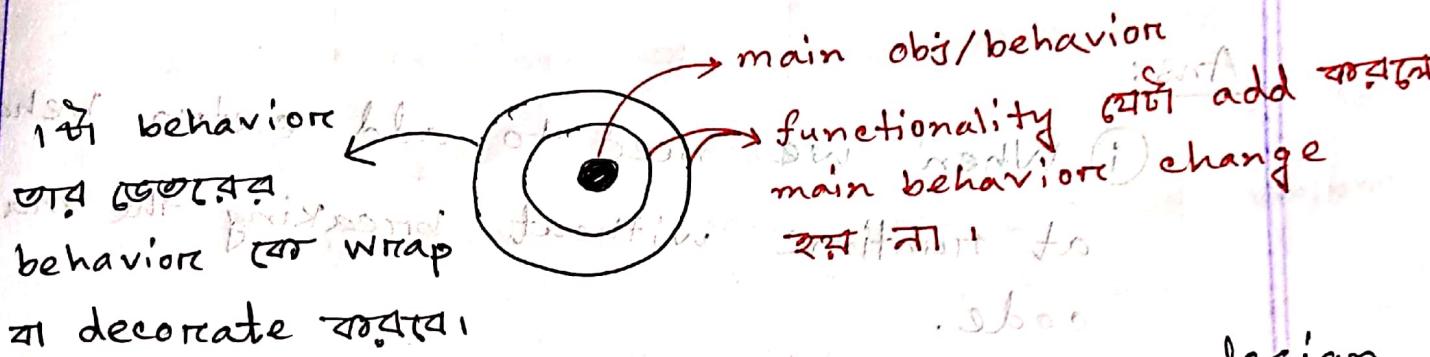
[~~IFLYBEHAVIOR~~ fly behav.
go fly() metho
d]

[same for Quack]

Generalised UML



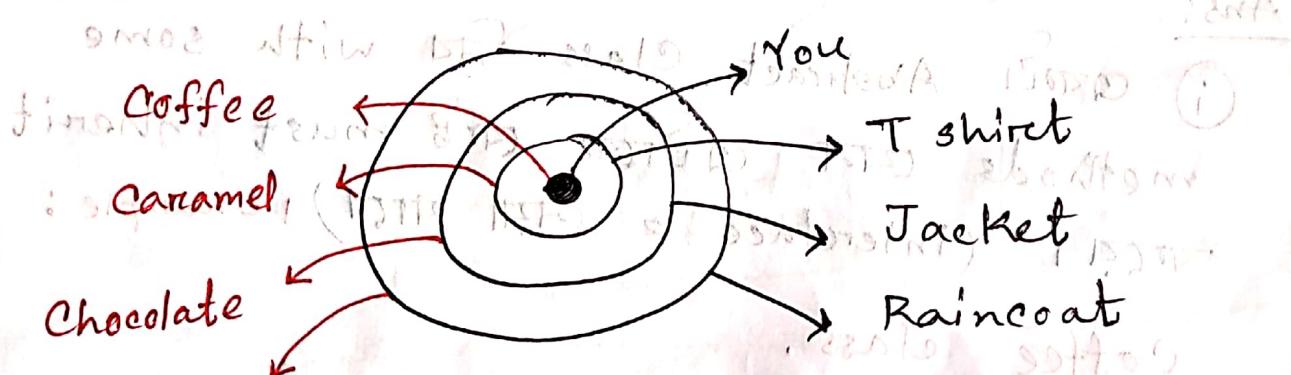
2. Decorator Pattern



Definition: Decorator pattern is a design

A ~~pattern~~ pattern that allows behaviors ~~to be~~ to be added to an object, dynamically, without affecting the object.

Real life example:



Flavor

* Abstract class এর child থাক্যা must। এবং child এর non-abstract class হলে অক্ষত abstract method এর মুলে override করবে।

Q: - When/At which situation should we use decorator pattern?

Ans:

- ① When we need to add "extra behavior" at runtime without breaking the main code.
- ② এখন inheritance এর difficult or unefficient, তাহলে It provides a flexible alternative to subclassing.
(When we need to extend behavior).

Q: ফিল্টার implement করবে?

Ans:

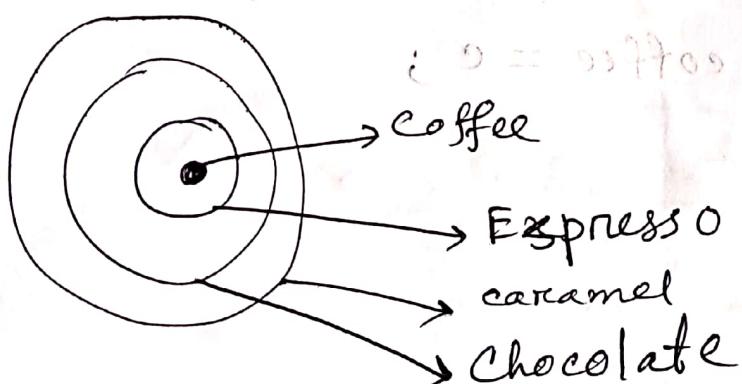
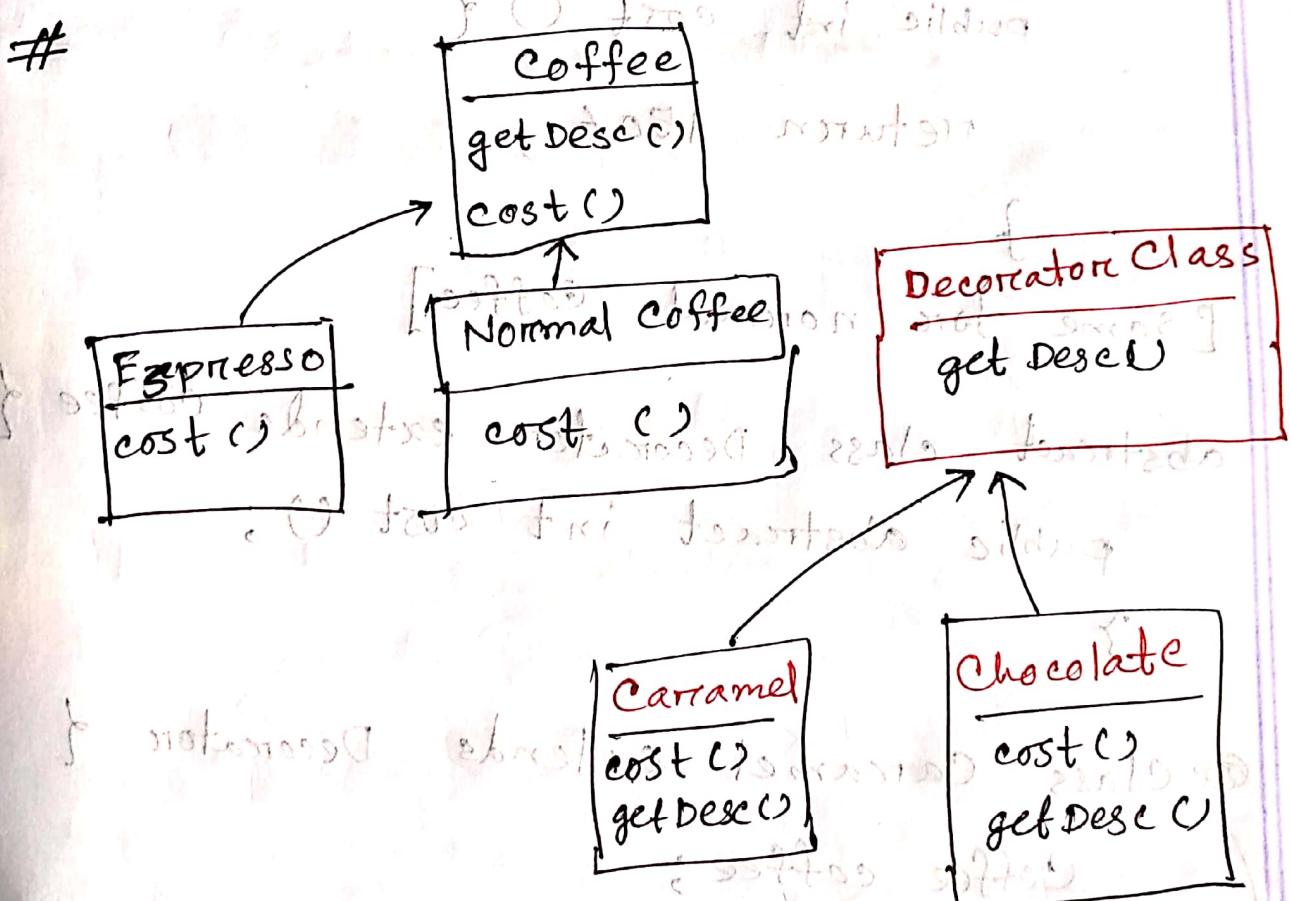
- ① একটা Abstract Class নিয়ে with some methods যেন এটাকে কোটি must inherit করবে। (interface ও নেওয়া শব্দ)। Example:
Coffee class.
- ② এখন filer class এর class কে inherit করবে এবং abstract method থাকলে override করবে। Ex: Espresso, Normal coffee.

[এগুলোকে concrete class ও বলে]

Abstract

iii) একটি Decorator class নিয়ে, যার ~~কিছু~~ sub classes হাবলে এবং ওই class প্রস্তাব মধ্যে সম্পর্ক (Runetime) এ choose করা যাবে।

[Decorators] class is a type of main class and also has a main class [coffee] component.



#Code:

```
abstract class Coffee {
```

```
    public abstract int cost();
```

```
Class Espresso extends Coffee {
```

```
    public int cost() {
```

```
        return 150;
```

```
}
```

[Same for normal coffee]

```
abstract class Decorator extends Coffee {
```

```
    public abstract int cost();
```

```
}
```

```
Class Caramel extends Decorator {
```

```
    Coffee coffee;
```

```
    public Caramel(Coffee c) {
```

```
        this.coffee = c;
```

```
}
```

has
coffee

public int Cost {

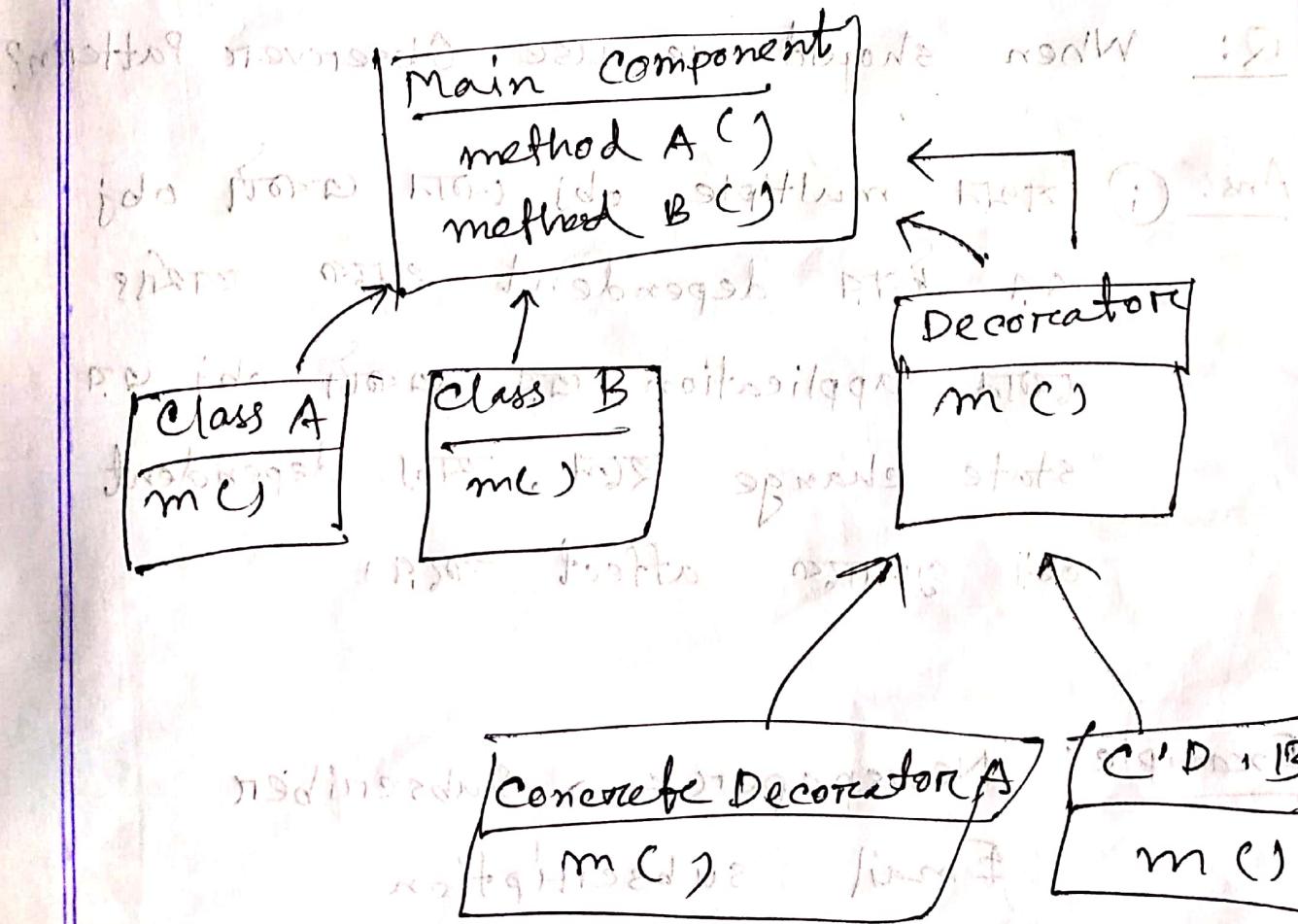
 return 20 + this.coffee.cost();
}

[same for chocolate]

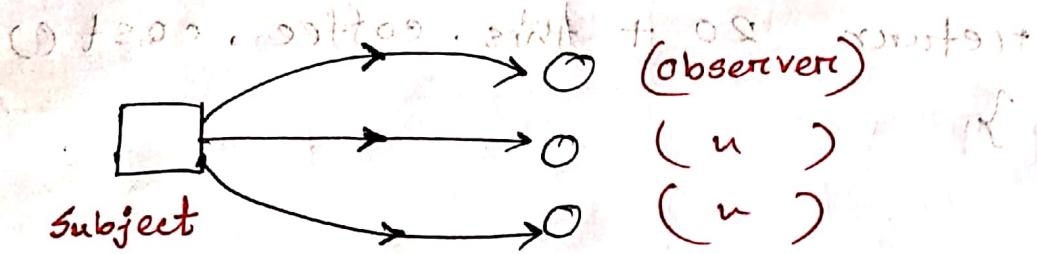
Espresso + Caramel

coffeeC = new Caramel(new Espresso);
coffeeC.cost() // 150+20 = 170

UML



3. Observer Pattern



Definition: Observer pattern defines a one to many relationship, where one object change will affect dependent objects.

(কে notify করা হবে)

Q: When should we use Observer pattern?

Ans: i) যখন multiple obj রেখার একটি obj এর উপর dependent হাতে অভিযোগ কোর্তা application এর একটি obj এর state change হলে, অন্য dependent obj খুজাতে affect করে।

Example: Newspaper ↔ Subscribers
Email subscription

ଯାଏ observe କରିବା ହେଲା → Subject

Q: କିମ୍ବା କିମ୍ବା implement କରିବା ?

Ans: ① ଏକଟି interfaceଟିକିମ୍ବା subject ହେଲା ତା
ଏବଂ ସମ୍ଭବ ନାହିଁ notify, add, remove ମୁକ୍ତିକାରୀ
କରିବାକୁ

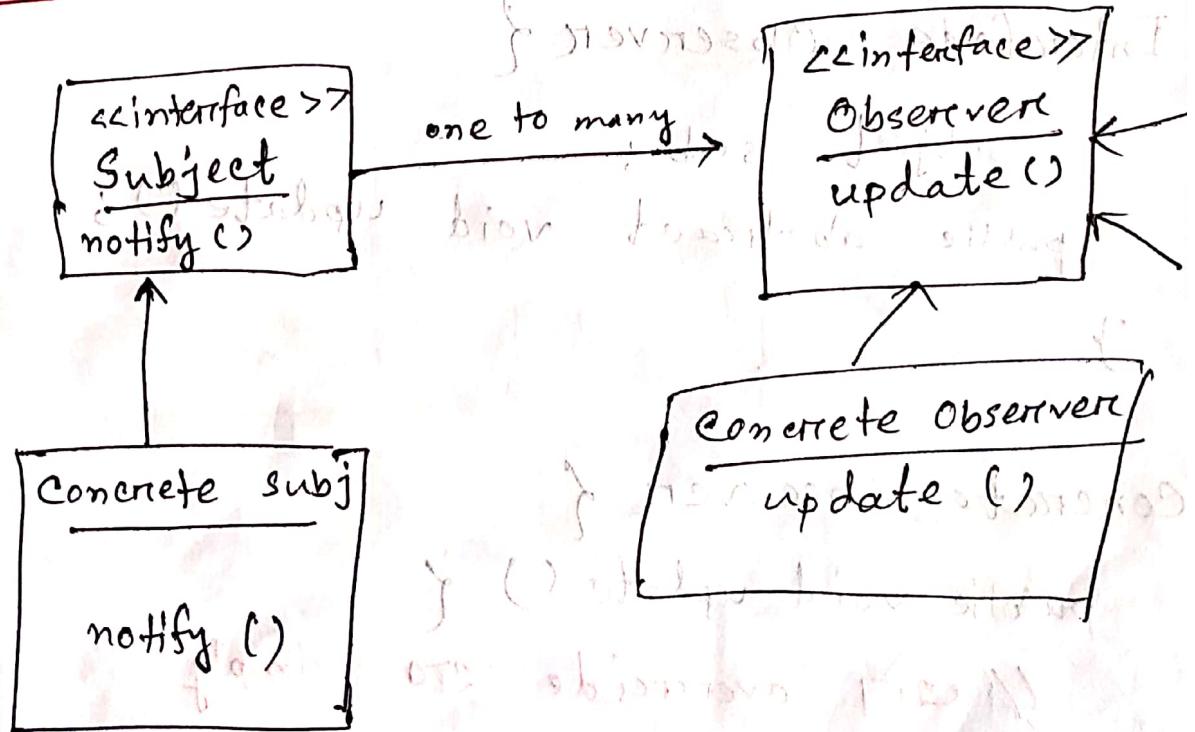
② Observer ଏବଂ ଏକଟି interface ହେଲା

ଏବଂ ଏକଟି subclass (concrete observer) ହେଲା
କିମ୍ବା update() method କିମ୍ବା
କିମ୍ବା ଏକା ଫଳାଫଳରେ update() method କିମ୍ବା

③ Subject ଏବଂ ଏକଟି concrete class
କିମ୍ବା ଏବଂ ଏକଟି ଫଳାଫଳରେ
notify, add, remove ମୁକ୍ତିକାରୀ
method କୁଣ୍ଡଳୀ implement କରିବା ହେଲା

ଯାଏ
observer
ଏବଂ array
ଏବଂ କୋମଳ
subject
ଏବଂ
subclass
କରିବା
ହେଲା
କିମ୍ବା

UML



~~Code~~

```

Interface Subject {
    List<Observer> obs = new ArrayList<Observer>();
    void add(Observer o) { ... }
    void remove(Observer o) { ... }
    public void notifyObservers() {
        for(Observer obs : observers) { // Array iterate
            obs.update(); // করে সবাই update() call দেন।
        }
    }
}
  
```

Interface Observer {

```

Subject sub;
public abstract void update();
}
  
```

class Concrete_Observer {

```

public void update() {
    // এটি override রে সিম্পল।
}
  
```

}

4] Template Method Pattern

Definition:

The template method pattern defines the skeleton of an algorithm in superclass but subclasses can override specific steps/ methods without changing its structure.

It helps to avoid code duplication.

Q: When to use Template Method Pattern?

A: i) When several ~~or~~ classes contain

almost identical algorithms with some minor differences.

template method converts a common

pattern to a superclass & adds difference

to subclass.

ii) When client to particular step overrides

to extend behavior but same algorithm

to do

to do

১টি Algo এর steps গুলো method আবারে থাকবে,

~~Explain~~ ১টি Algo এর steps গুলো method আবারে থাকবে।

Q: কিভাবে implement করব?

১) ১টি abstract class থাকবে যার মধ্যে

steps আবারে method থাকবে। যার

২) templateMethod() থাকবে যেটা

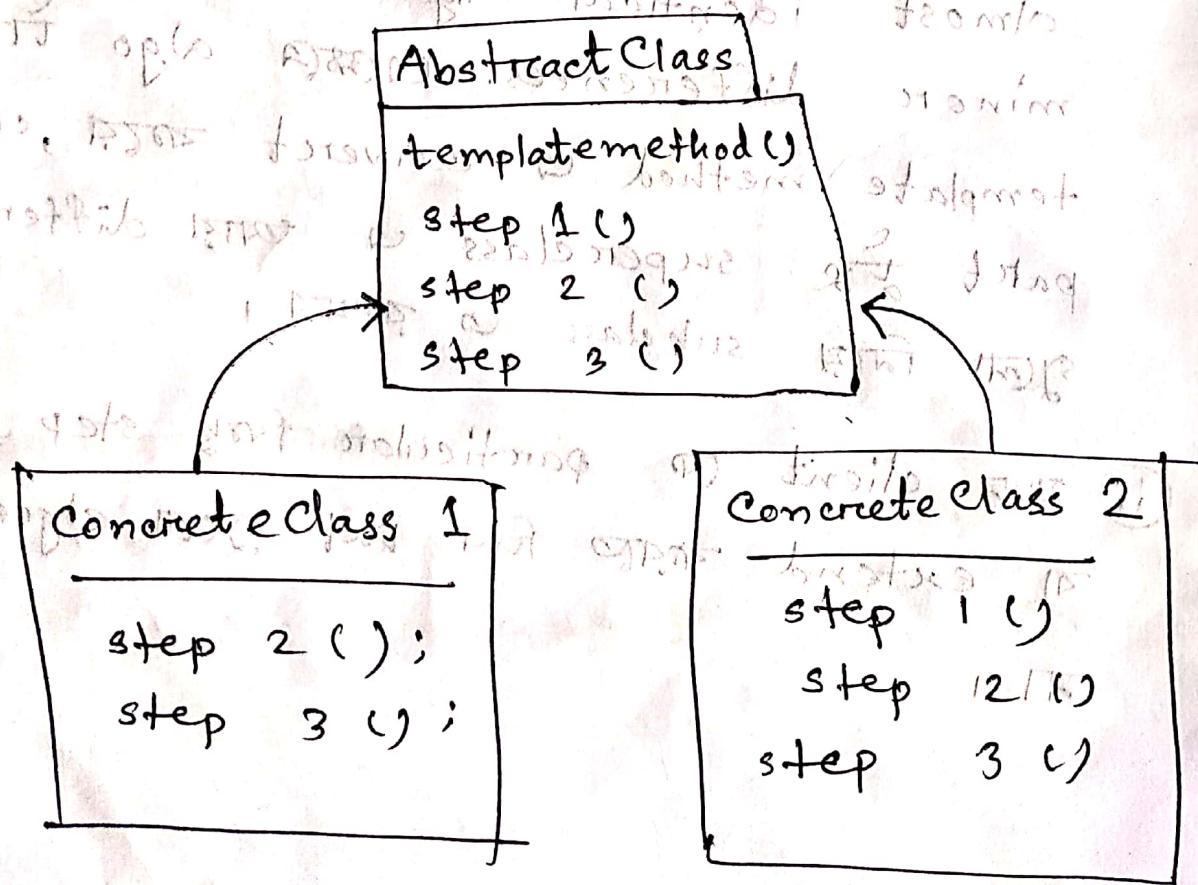
abstract/Non-abstract methods গুলো এক call করতে পারবে।

৩) Sub/concrete classes গুলো প্রয়োজন

অনুসারে দরকারি method গুলো override

করবে তবে কোথায় templateMethod()

করবে তবে কোথায় override করবে না।



~~Code of abstract class~~ ~~in standard C++~~

```
abstract class JuiceMaker {  
public:  
    void addWater(); // abstract method  
    abstract void makeJuice(); // Template class  
    void serve();  
    void step1(), step2(), ...  
};
```

Mango juice extends JuiceMaker {

```
class MangoJuice extends JuiceMaker {  
public:  
    void makeJuice() {  
        step1();  
        step2();  
        add sugar();  
        blend();  
        serve();  
    }  
}
```

Orange juice extends JuiceMaker {

```
class OrangeJuice extends JuiceMaker {  
public:  
    void makeJuice() {  
        step1();  
        blend();  
        serve();  
    }  
}
```

Q: what is hook in template pattern?

A: An optional step / method, একটি

override করা যাবে। নতুন subclass

চাইলে additional কাজের জন্য একে

override করতে পারে।

(i) constructor
(ii) deposit
(iii) withdraw
(iv) getBalance
(v) transfer

একাধিক template method পাবলি possible.

5 | Prototype Pattern

Definition: Prototype pattern এমন এক design

pattern যেটা use করে duplicate object create
করা যায়, scratch থেকে obj create বহুল
ব্যবহৃত।

Q: কখন use বহুল?

① যখন scratch থেকে obj creation costly
ও time consuming.

② এই class এর obj copy করব সেটার উপর
dependent হলে যদি অসম্ভব হয়।
pattern কখন use বহুল যাবে।

Q: কিভাবে implement বহুল?

Ans: ① ~~একটি interface~~ ~~নিয়ে prototype~~

② এক abstract class নিয়ে যেটা cloneable
কর implement বহুলে।

③ এক abstract class কে extend করব
এমন একটি concrete class নিতে পারি।

এখন এই concrete class গুলি copy করা সুজ্ঞত
পার।

III

এরপৰ client যেনা ক্লোন obj দৰকার

ইল এ মেটা copy/clone ~~কৰা~~ কৰে এন
use কৰিব।

গোপনীয় → Exam
শ্রাবণ, part A, 2(a)

Fahim → note

Iterator Pattern

→ কিৰি তিৰ্য্য type দৰিবজ্যে কৰা কৰা iterate

বৰাবৰ পাখে কৰা কৰা কৰা



function

→ একটা interface use কৰে diff type

obj কৰি access কৰত পাৰি,

→ Example:

Football team → List এক এক
baatch পৰিবেশ কৰিব (Array, Linkedlist,
HashMap)...

→ Advantage: Simplified, both side (one
traverse কৰত যাব,

Adapter Pattern

#code:

```
Abstract class Car {
    int weight_kg;
    int mileage_km;
    abstract void start();
}
```

```
Abstract class Bike {
    int weight_pound;
    int mileage_mile;
    abstract void StartofBike();
}
```

Desire for bike establishes level with ←
 class BikeToCarAdapter extends Car {
 @override
 void start() {
 bike.StartofBike(); // Bike to method set
 }
}

car through access to
 Bike.start();

Class Adapter main {
 public

Car car = new Car();

Car bike2 = new Bike2CarAdapter(New Bike);

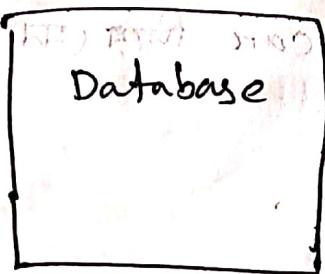
bike2.start(); // StartofBike run 250
 actually -

}

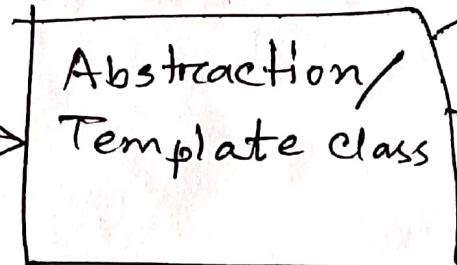
}

Dependency Inversion Principle:

⇒ High level modules should not directly depend on low level modules.



High level
modules



Abstraction
class

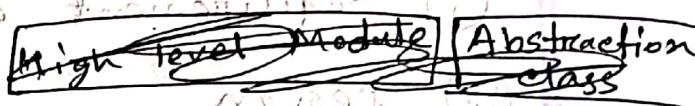
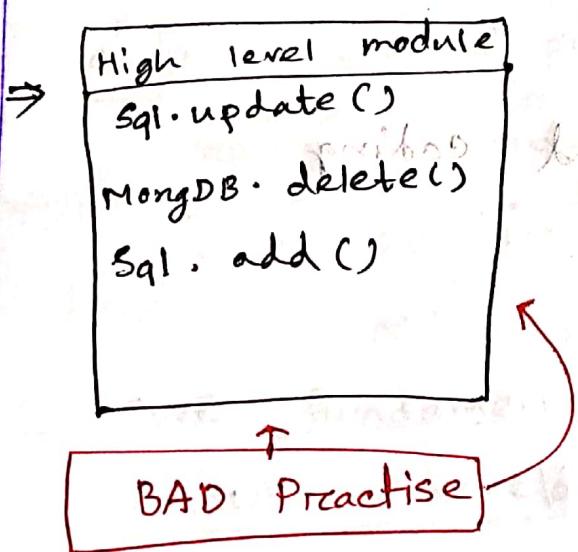
SQL

Mongo DB

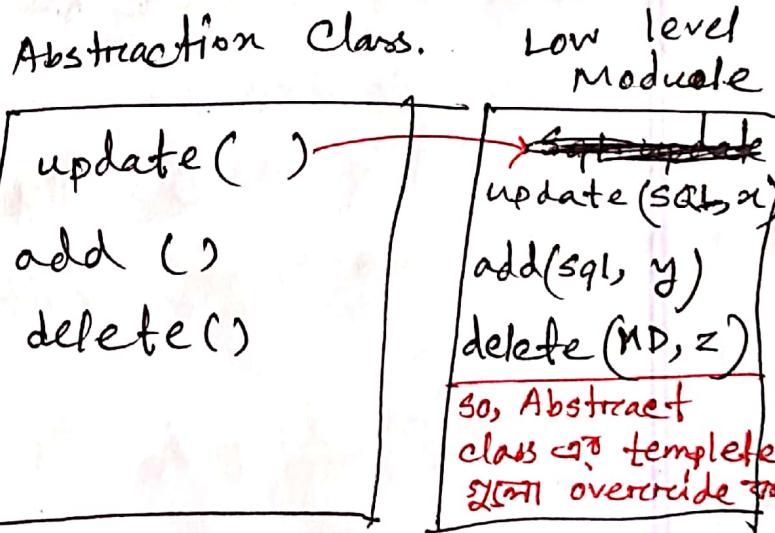
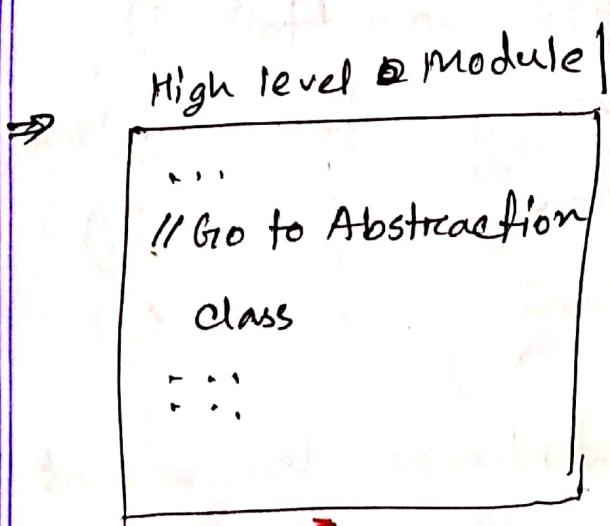
Low level
modules

⇒ So, High level modules এর মাধ্যমে প্রকৃতি
abstraction এর মাধ্যমে Low Level
module এর মাধ্যমে interact করতে হবে।

⇒ Sir এর মতো, open close principle এর অন্তর্ভুক্ত আসলে D.I. Principle



Tightly coupled এবং প্রোগ্রাম কোডের স্থান পরিবর্তন করা কঠিন।
add / remove করা কঠিন।



Good Practise