#10

Design Patterns

General ~~repeatable~~ repeatable solutions to commonly occurring problems.

→ ⊛ Why do we need design patterns?

- Shows relationships and interactions between classes and objects
- Speeds up development
- Reusable code
- Maintainable and flexible code
- Readable code

Categories → 1) Creational
2) Behavioural
3) Structural

OOP Concepts
[Design Patterns]

05-01-22

OOP → Object Oriented Principles
- Principle for design and development of programs using modular approach
- Organizes code in objects

Object Instantiation in Java

Employee employee = new Employee(); → [arguments passed to constructors]

- Employee → Object Class
- employee → Object name
- new → new memory allocation
- Employee() → Class as a function

Doing this part gives the object reference only, but allocates no memory.

Declares | Allocates

(*) Inside a class, to distinguish between variables and methods of a class with variables and methods of other classes, we can use this keyword.

Types of Methods

→ Modifiers [Ex: setters]
→ Accessors [Ex: getters]
→ Constructors → Called when an object is created. Initializes values of the variables
→ Destructors

Access Modifies

→ Public, protected, private

OOP → encapsulation, abstraction, inheritance, polymorphism.

Inheritance in Java

```
class Executive extends Employee{

}
```

in C++

```
class Executive: Employee{

}
```

You can assign ~~the~~ an object of a child class inside a parent class.

Ex
```
class Dog {----}
class Poodle extends Dog {----}
Dog myDog = new Dog();
Poodle myPoodle = new Poodle();
myDog = myPoodle;      // legal
myPoodle = myDog;      // illegal
myPoodle = (Poodle) myDog;  // still illegal
```

Polymorphism

→ Ability of a function to take multiple forms.

[Method Overriding/Loading → same name, different parameters (type or no.)

Method Overriding

↳ same function signature (same name and parameters but different body)

How to use overridden methods?

```
class Family extends Person{
    void Birthday(){
        super.birthday(); // call overridden method
    }
```

super. → New Keyword

Encapsulation

→ Hiding data from others, and giving access to others in a restricted format.

Abstraction

↳ Hiding implementation details from users, and showing functionality only

Abstraction → Interface → has abstract methods only
→ Abstract Class → has abstract, non-abstract method
[abstract keyword] ↓ method with no implementation

→ Why do we need abstract class?

Subclasses implement the class in their own way. (Code Reusability)
Parent class will contain the common behaviours.
Subclasses are FORCED TO implement the abstract method.

Interface can NOT BE extended (like abstract classes) but they are implemented [Keyword]

→ Why do we need Interface?

Multiple Interfaces can be inherited by ~~multiple~~ a single classes,
but abstract classes do not allow multiple inheritance

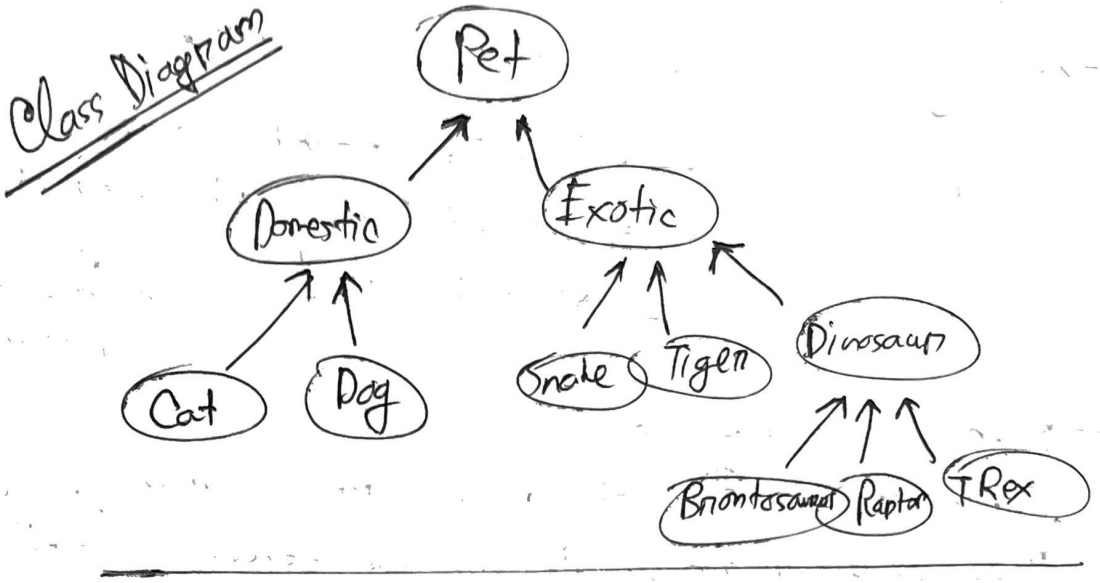
class C extends A, B {---} ✗ Wrong
class C implements A, B {---} ✓ Right

Ex - (20 behaviours/class) Subclass 1:10. [19 behaviours are common] → (What should you use here?)

As 19 behaviours are common, we can write them in a single ABSTRACT class and leave 1 abstract method for the unique behaviour.
(Ans.)

Ex -

Class Diagram



(i) main {

Dog d1 = new ~~dog~~ Dog ("Fido", 10, "Phil");

Snake s1 = new Snake ("Snaggles", 7, "Katie");

Exotic reference;

reference = d1;

System.out.println(reference);

}

Output:

~~Fido the dog~~

~~Fido~~ ~~Fido the~~

~~10~~ ~~cutenessFactor = 8~~

~~Phil~~

Error

Parent class of dog is unrelated to exotic.

(ii) main {

Cat c1 = new Cat(.....);

Cat c2 = new Cat(.....);

Trex trex = new Trex(.....);

Tiger tgr = new Tgr(.....);

c1.setCutenessFactor(9);

tgr.setDangerFactor(2);

} → Output if we print all 4 classes.

(*) If you don't call a method and try to print the object → the toString() method is called.

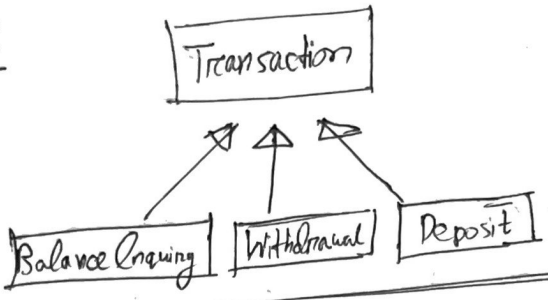
Output: I am Princess the cat and have a cuteness factor of 9.
I am Issy the cat and have a " " " 7.
I am Fancy Pants the TRex " " " larger " " 10.
Tiny the tiger " " " larger " " 2.

H.W. exercises
change of design → class diagram
[how design in changed due to inheritance]

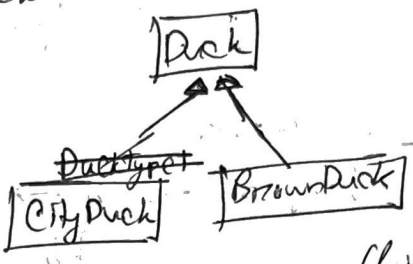
#3

Design Patterns

Solution of H.W.



Problem → Different type of duck → soln. → inheritance

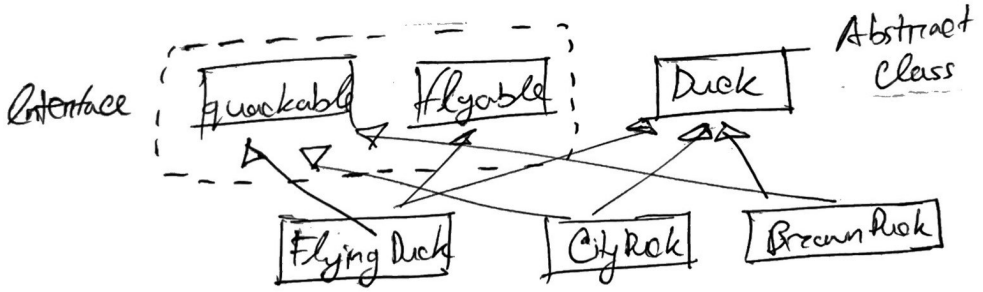


Problem → implement a new type → flying duck. with fly method

⊛ [some subclasses need this method, others don't].

soln.

Decouple behaviours from parent class as an interface.



Design Principle-1: Identify the aspects of your application that vary, encapsulate them so it won't affect rest of your code, and seperate them from what stays the same.

is a relationship → inheritance

has a " → composition

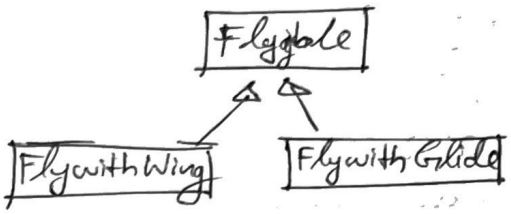
Design Principle-2: Program to the interface, not to the implementation.

Design Principle-3: Favor composition over inheritance.

↓

As part of a class

What if there are multiple ways to fly?



Example of programming to interface:

```
Dog d = new Dog();          ✗ Implementation
d.bark();

Animal ani = new Dog();     } ✓ Interface
ani.makeSound();

ani
a = get Animal();   ⟶ To know which animal it is
a.makeSound();
```

Example of composition over inheritance:

```
X public class CityDuck extends Duck implements Quackable {
                                  ↓
                          inheritance X
}
public class CityDuck extends Duck {
    public MallardDuck() {
        quackBehaviour = new Quack();
        flyBehaviour = new FlywithWings();   } composition in constructor
}
```

```
Class Duck {
        FlyBehaviour flyb;
        QuackBehaviour quackb;
        performFly(){
            flyb.fly();
        }
        performQuack(){
            quackb.quack();
        }
}
Client {
    Duck mallard = new MallardDuck();
     mallard.performFly();
     mallard.performQuack();
```

Duck has a flybehaviour.

→ use composition.

But Mallard duck is a duck.

→ use inheritance

```
}
```

What if we modify the duck class?

// inside the duck class

```
setFlyBehaviour (FlyBehaviour fb){
        flyBehaviour = fb
}
setQuackBehaviour (QuackBehaviour qb){
        quackBehaviour = qb
}
```

} Can also be set dynamically during runtime → Context class (see slides)

// inside the client class

```
mallard.setFlyBehaviour (new FlyWithRocket());
mallard.performFly();
```

⇓

→ Strategic Design Pattern

#4

Template Pattern

Design Pattern

- Creational
- Structural
- Behavioural – [Strategy]

Template Pattern

* Template Method → calls all the steps in a class
  * make it final to prevent overriding

```
template_method(){
    step1();
    step2();
    ...
}
```

* Create an abstract class which is the template with a template methods that calls all the steps/methods for a process.

(parent; skeleton)

```
game(){                          → abstract class
    initialize()   }
    start()        }  common methods
    end()          }  [can be defined as well]
    play(){
        initialize()   }
        start()        }  Template method → make it 'final'
        end()          }
        hook()         }
    }
```

So, child classes don't override it.

(next page)

Problem Solved by Template Pattern

→ Keeps the common properties/methods in a single class.

Use Case

→ For Common Processes [sets of steps]

algorith → series of steps → steps into methods → methods inside a single template method

Q. What if the game has an additional step after the game ends?

(*) Hook method () → An implemented method that does nothing

```
void hook(){
}
```
→ Child classes can override this method to add additional steps/methods.

(*) Remember hook() is not abstract as overriding is optional.

⇒ The sequence of steps CAN NOT be altered in template pattern, but additional steps can be added.

⇒ The original steps will however vary in subclasses.

Design Principle → Hollywood Principle

Don't call us, we will call you.

↳ Prevents dependency rot.

↳ when high-level components depends on low-level components.

↓

High level components will call low-level components.

by declaring parent classes ~~Template~~ with a template method (high-level) which calls the ~~other~~ (low-level) components.

other methods

Template class also ensures encapsulation.

(steps of the process is hidden from client)

Qn

(#) → if you want the method implemented forcefully, make it abstract

→ if you don't want that, make it `void nameMethod(){ }`

Comparison → Singleton vs Prototype

~~Def Syllabus~~

Adapter Patterns Again

Dependency inversion principle

↳ High level modules should not depend on low level modules; both should depend on abstraction.

↳ Abstractions should not depend on details. Details should depend upon abstractions.

Q: How adapter follows dependency inversion principle?

Implementation → inheritance from abstract class.

creating containers for abstract class.

i.e. - do not DIRECTLY create objects of lower level/child class.

→ Object creation is a low-level task and users should not depend on them.

Post Mid

Observer Pattern

→ Subscription/Notification ~~type~~ process → Behavioral pattern

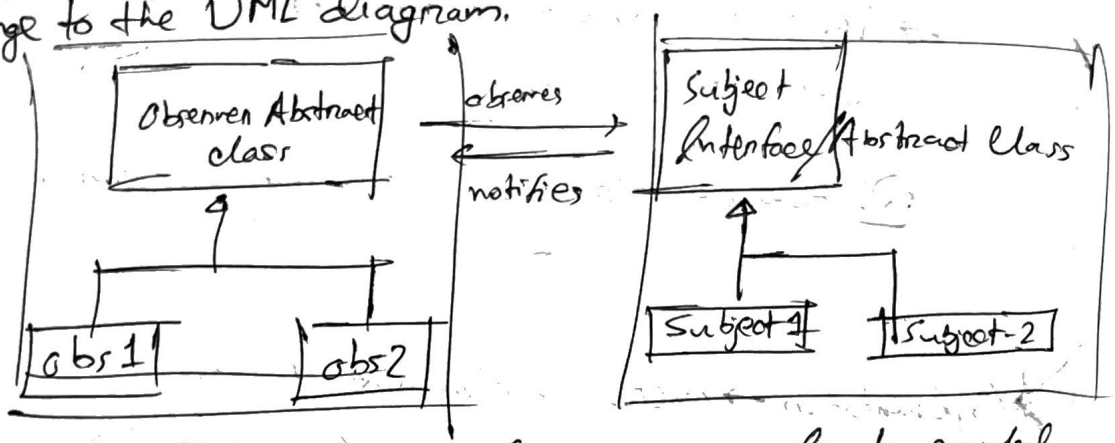
When it is used:

→ One to many relationship with a publisher ~~relationship with~~ (notifier)

Implementation:

→ * There is a publisher/subject who keeps track of a list of observers and ~~sends~~ sends notification when the state is changed.

→ A change to the UML diagram.



Important

* Observer Pattern → subjects and observers are loosely coupled

why?

→ If there is a change then ~~observer~~ subject notifies the observer. ~~Subject~~ Observer is not dependent on the ~~observer~~ subject.

Similarly, subject is not dependent on observer either.

Hence, loosely coupled subject and observer.

~~Advantages~~ →

Subject and observer communicate with one function

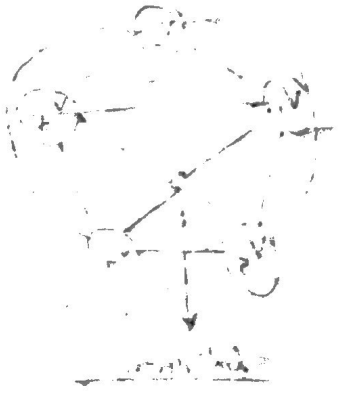
Mediator Pattern

→ Behavioral Pattern → Many-to-many relationship. Ex - chatRoom

(*) Reduces communication complexity by creating an intermediary object.

→ Objects are not dependent on each other but dependent on intermediator class → [loosely coupling] (*)

Observer vs Mediator	
→ Notification/ Subscription	→ Communication between multiple objects
→ 1 to Many	→ Many to many



Builder

→ Creational design pattern

→ Combines different types of object into a new complex object.

Problem → create a complex object

[constructor will have multiple parameters]

Factory vs Builder

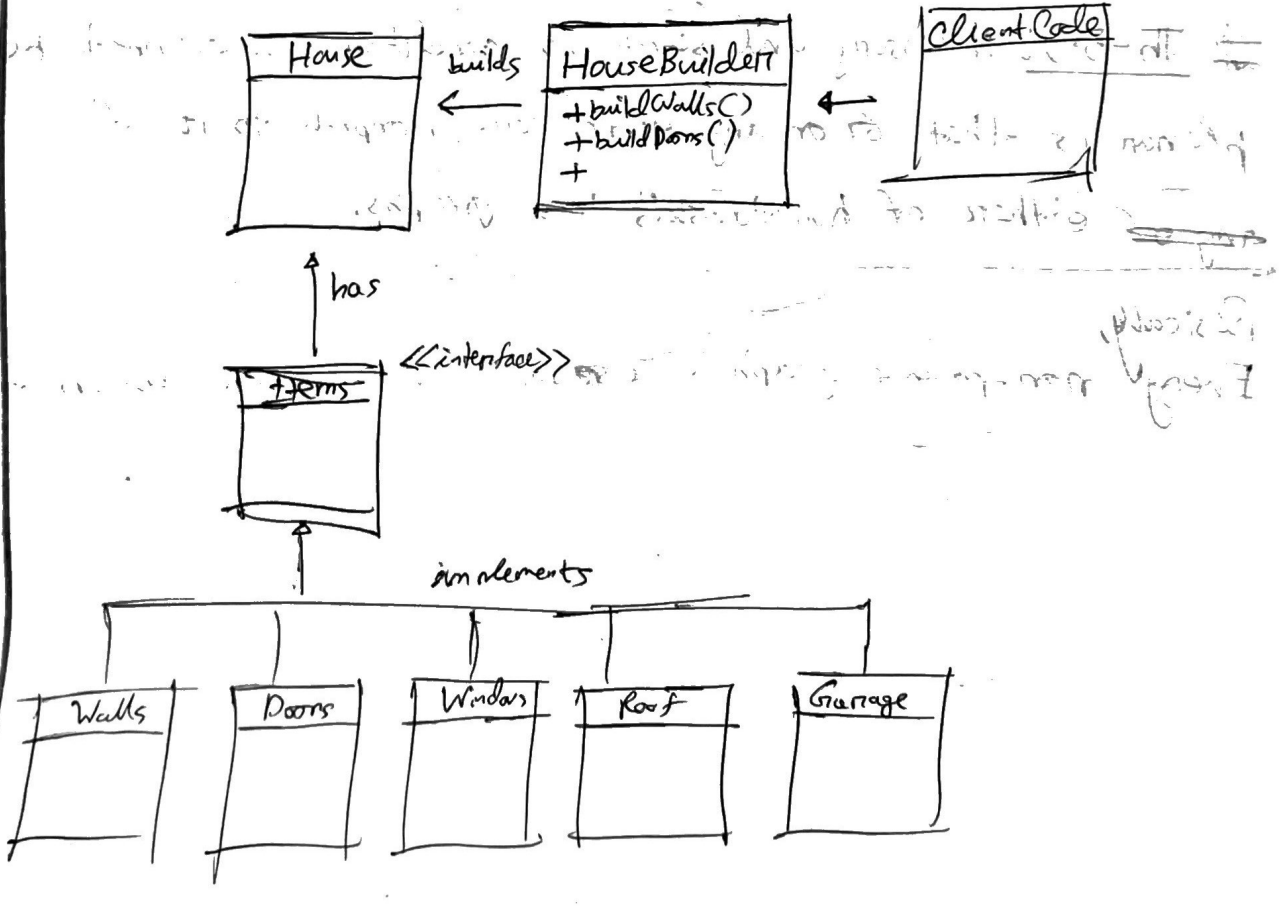
↳ Seperates object creation for a single object

↳ Seperates Object creation for a complex object

House Builder

Extracts the object creation code out of its own class and move it to seperable objects called builders.

```
HouseBuilder houseBuilder = new HouseBuilder();
House house = houseBuilder.buildWalls();
```



Command Design Pattern

→ Behavioural pattern

[Request is wrapped as a command object] (I don't want to expose operations to my client).

Mediator vs Command

Many to many relationship notifications	Want one class to do something client will not know about the command.

Proxy [Structural]

→ A class that represents functionality of another class

→ Reduces the memory footprint (caching)

→ store locally to reduce access to memory

→ Hiding real object from client

[Ensures security]

Memento [Behavioural]

→ Restore state of an object to previous state

Similar to version control

Proxy vs Memento

caching	store previous states of objects

Composite [Structural]

→ create a group of objects and treat them a single object.

→ use ~~object~~ when objects have a tree structure hierarchy.

Complex Object → Multiple Objects Grouped

(*) Maintaing a single object is easier than maintaining multiple objects.

→ Single Object and complex objects are treated in the same way.

Assignment-2

Composite Pattern to implement a scenario.

[আতিক er copy থেকে cross-check করবো।]

Composite	vs Builder
structural	creational
→ getting different structure of object (complex object structure) and we are creating a complex object.	→ seperate creation logic and creating complex object in a seperate method.
→ tree-like structure objects	

Bridge (similar to strategy)

→ Memorize slides

→ splits a large class into two hierarchies – abstraction, implementation

(#) Seperate the behaviours ~~from~~

Maintaining abstraction and encapsulation

because changing a part will not affect elsewhere.

We are preferring composition over inheritance.

Facade

→ Hide complexity [similar to proxy]

Why abstract factory is creational but facade is structural?

↳ deals with object creation only

↓ defines some methods that ~~form~~ form relationship

As we deal with calling the method, it is structural.

Flyweight

→ Using already created objects

Similar to prototype but we will NOT clone/copy. We will reuse the already created objects (not clone it), create new object if that previous object doesn't exist.

→ used mostly in game development.

Code Smell → Common occuring problem in code called ~~smell~~ smell.

Long Method → Method is too long to comprehend

1) → extract method → replace them with query → introduce parameter object

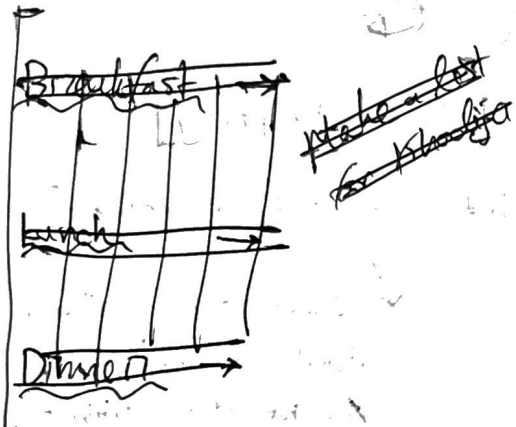
2) → replace method with object method → 3) don't do it inside, do it outside → 4) Decomposition if else

Remember these 2 things — Check Atiq's Notes

i) instant method()
ii) encapsulation method()

→ Delete dead code

* Form Template Method



Important

Question in exam → identify the code smell and then fix it → more important — most of the marks

* Inside one scenario, there can be multiple code smells

* ~~Assignment Submission → Next Thursday~~

Code Smells

VV. Imp.

Switch Statement

→ Same if_else__ and switch statements are repeated.

↓ solution

for repeated if___else cases, we can make a class for a particular outcome.

→ or we can make a method and pass the conditions as parameters.

Introduce Null Objects

→ Using one if__else__ we can treat null objects.

(*) Relationship between refactoring and patterns.

Visitor Pattern

→ Visitor class changes the executing algorithm of an element.

Imp.

→ Difference between Visitor, Strategy, Decorator

Visitor (changes the steps of the algorithm) | Strategy (changes the algorithm)

Imp.

→ Read Double Dispatch (2-way communication)

Find out — Visitor vs Composite