Design Patterns

DATE / /
PAGE

- **Low level Design** is also known as object-level designing or micro-level designing

It involves converting high level design into more detailed blueprint, addressing specific algorithms, data structures and interfaces.

It act as a guide for efficient implementation of system's functionality.

- **High Level Design Vs Low Level Design**

It is general system design where we do trade off b/w diff frameworks, components, databases and we choose best considering business needs.	In LLD we define the components and how these components will be communicating with one another.

- **Design Pattern**

Design Pattern are reusable solutions to common problems encountered in software design.

They provide structured approch to design software by capturing best practices & proven solutions making it easier to develop scalable, maintainable and reusable software.

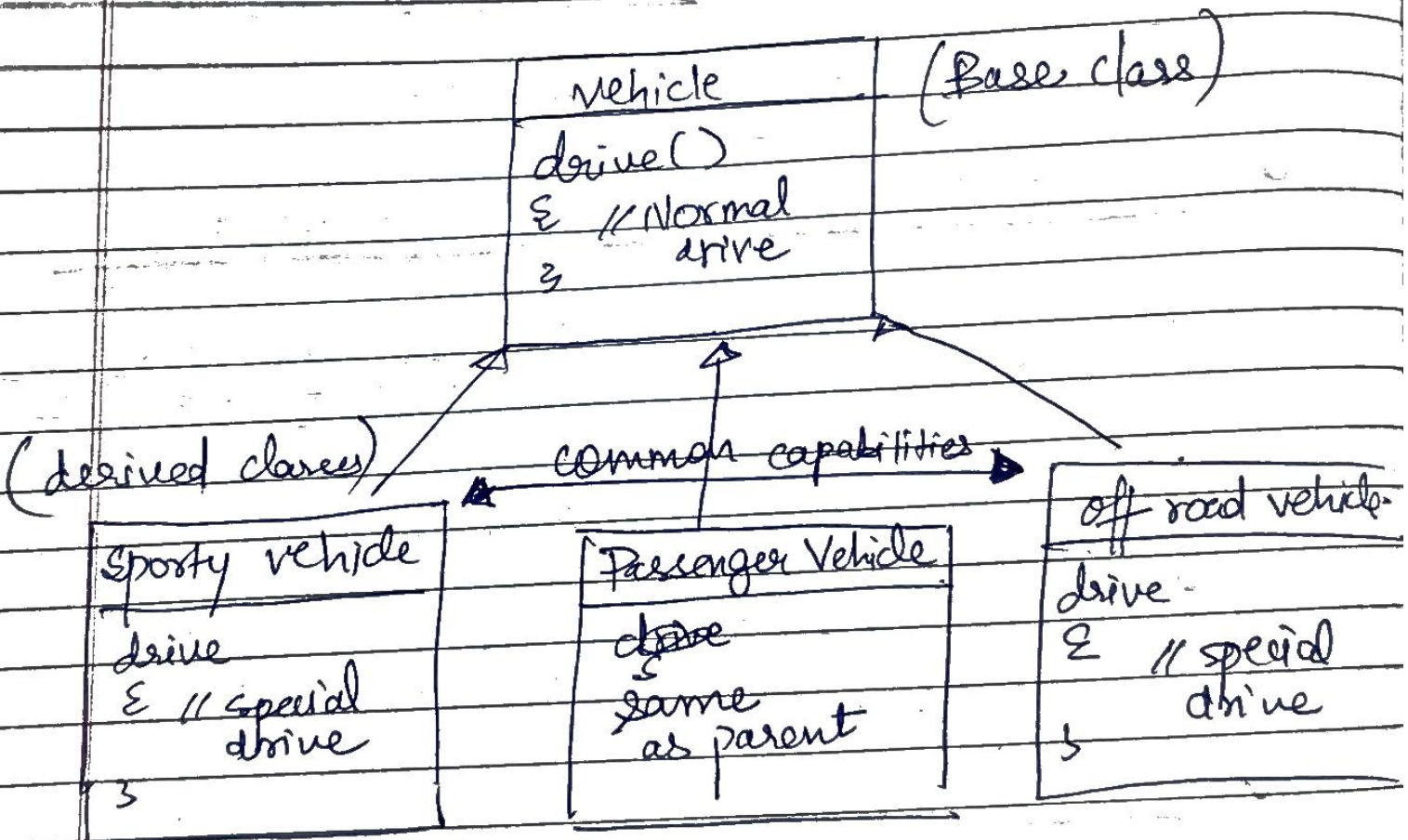
Scanned by Scanner Go

DATE : __/__/____
PAGE : _____

1. Factory Design Pattern

is-a ——→ has-a

1a: The digram below shows inheritence of different class from one base class vehicle



Let say the derived classes like sporty vehicle & off road vehicle has different implementation to drive function different from parent class. They override drive function.

~~Your classes are (derived) are~~

Both sporty vehicle and off road vehicle has same implementation of drive function

Scanned by Scanner Go

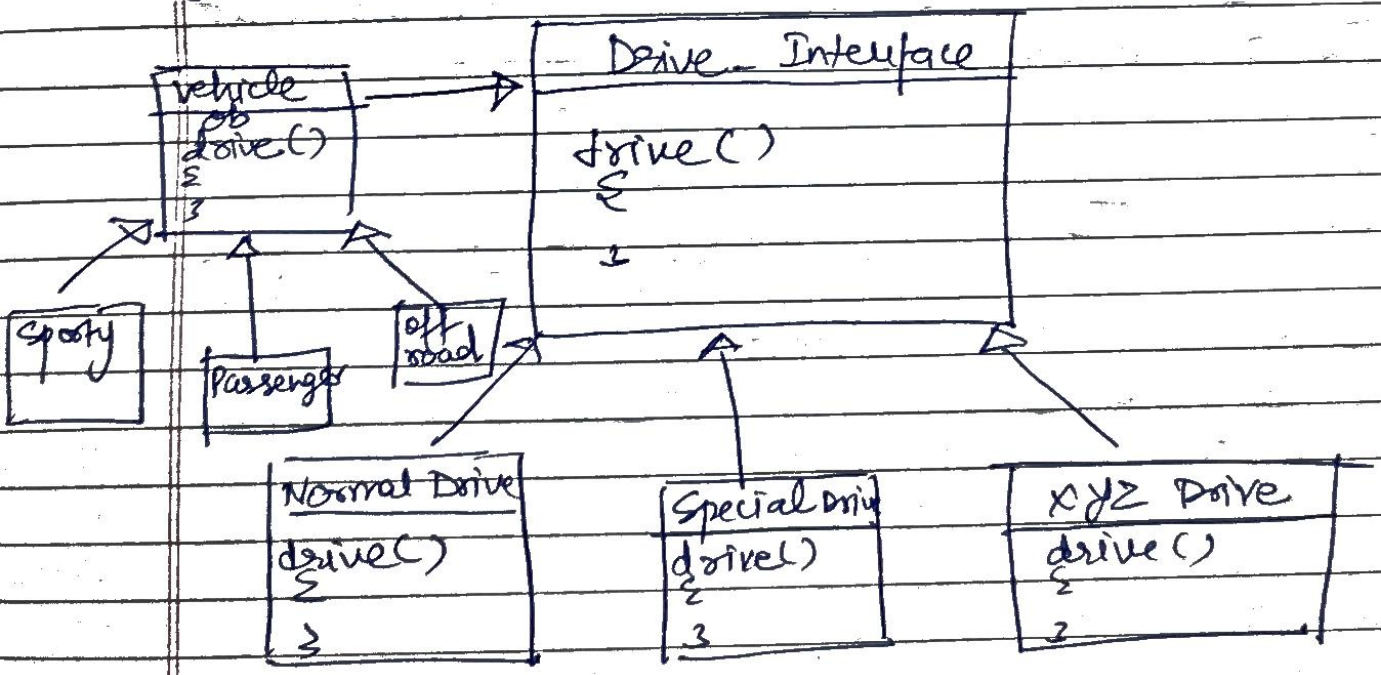
Problem: Code duplicacy in derived or child classes

Derived classes are not using parent class capability and the childs (derived classes) are having same capability & When the features of parent ↑ and system grows. This results in

- Code duplicacy
- low Maintainability
- less Scalability.

Solution via Strategy design Pattern

Here we create a drive Interface



Now Vehicle is implementing the the drive interface by creating object of drive interface.

Scanned by Scanner Go

DATE :__/__/__
PAGE :_____

And now child can implement any drive function and fully dependent on parent's capability instead of one another.

Now the system is scalable, reusable and maintainable.

2. Observer Design Pattern :-

This pattern defines one to many dependency between objects so that when one object changes state, all its dependent are notified and updated automatically.

Deals with interaction and communication btw objects

Components of Observor design Pattern :-

1. Subject :- The subject maintains a list of observers (listeners). It provides method to register and unregister observers dynamically and defines a method to notify observers of changes in its state.

Scanned by Scanner Go

2. Observer: It defines an interface with an update method that concrete observers must implement and ensures a common or consistent way for concrete observor to receive updates from the subject.

3. Concrete Subject :- Concrete subjects are specific implementation of the subject. They hold the actual state or data that observor wants to track.
When this state changes, concrete subject notify their observers.

4. Concrete observor - Concrete observers implements the observer interface. They register with concrete subject and react when notified of a state change.

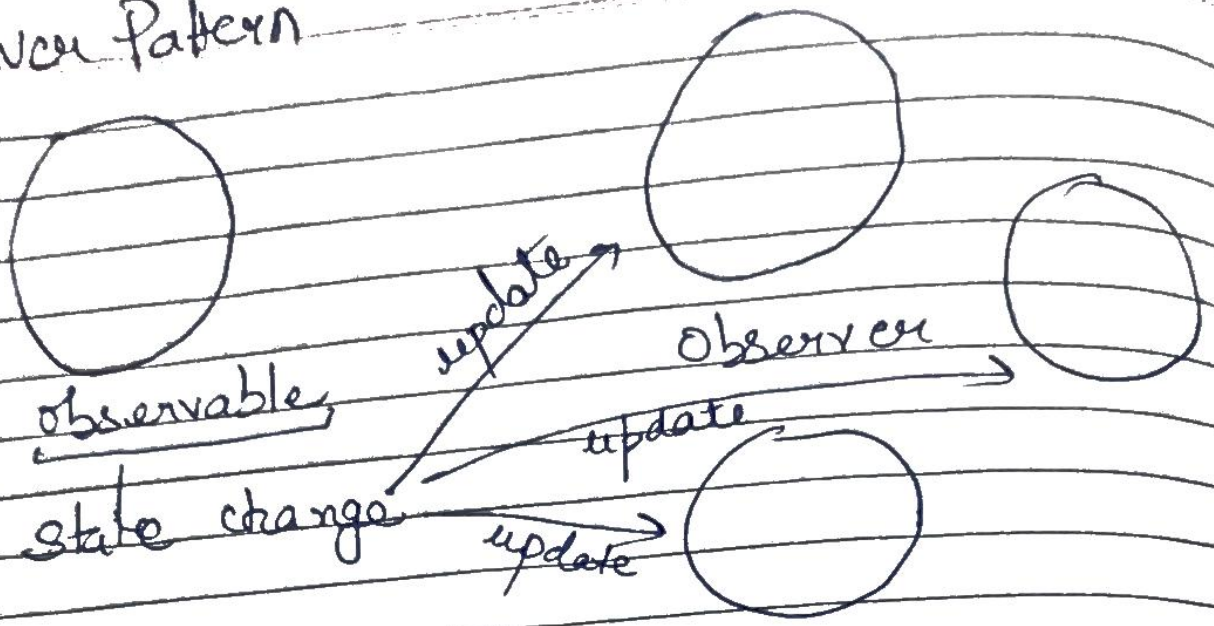
Question :

Amazon
- Search for iphone
- Iphone is out of stock
- So we click on **Notify me**

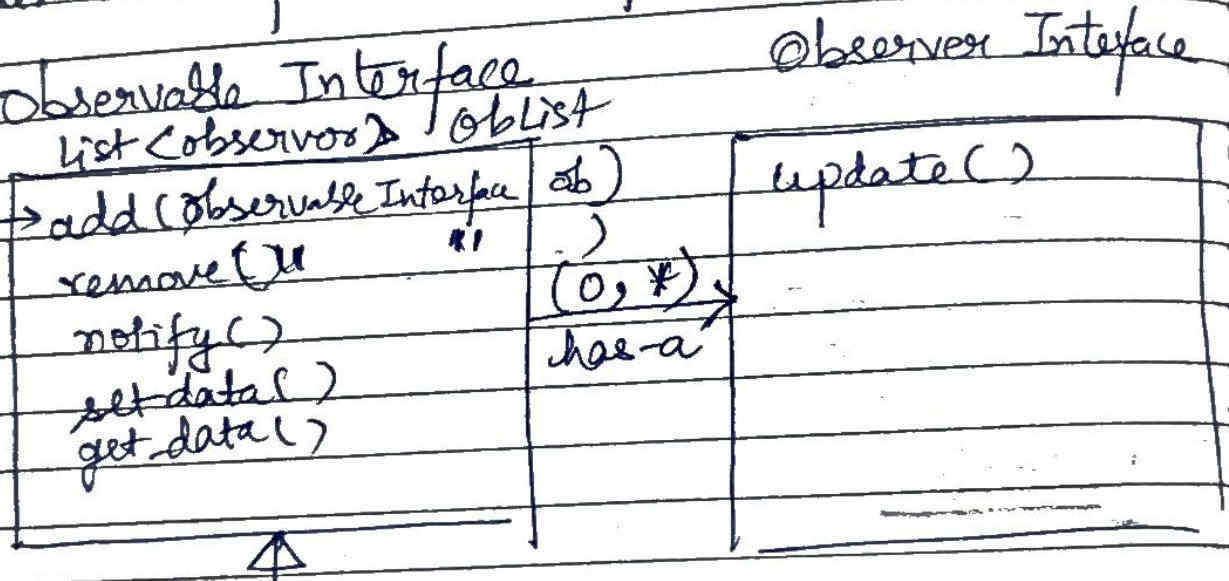
Implement this notify me.

Scanned by Scanner Go

Observer Pattern

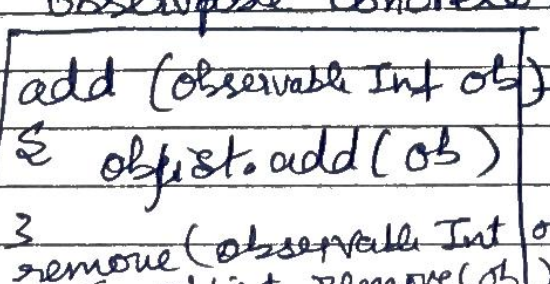


Implementation of observer pattern



List<Observers>
↳ contains the list of all the observers that depend on observable interface.

is-a



Scanned by Scanner Go

```
notify()
{  // just iterate for
all observer.
for(observer Int ob : objlist)
  {  obj.update();
  }
}

slt_data(int t)
{  data = t; // if anything changes
   notify() // we call notify
```

observable object

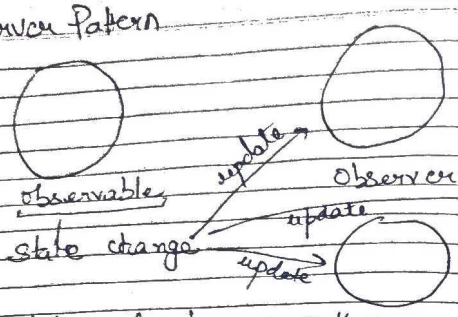
has a → Now no need to pass object

observer Concrete class

```
update(object o)
{  observable interf obj;
   update()
   {  obj.getdata();
   }
```

Scanned by Scanner Go

observer Pattern



Implementation of observer pattern

Observable Interface
List<observer> / obList

```
add(Observable Interface ob)
remove(" ")
notify()
set data()
get data()
```

(0, *) has-a

Observer Interface

```
update()
```

observer Concrete class

```
update(Observable Interface obj)
{ obj.get data();
}
```

has a → Now no need to pass object

```
notify()
{ // just iterate for all observer.
  for(observer Int ob : objlist)
  { obj.update( ) → observable object
  }
}

set data(int t)
{ data = t; // if anything changes
  notify() // we call notify
}
```

List<observer>
↳ contains the list of all the observers that depend on observable interface.

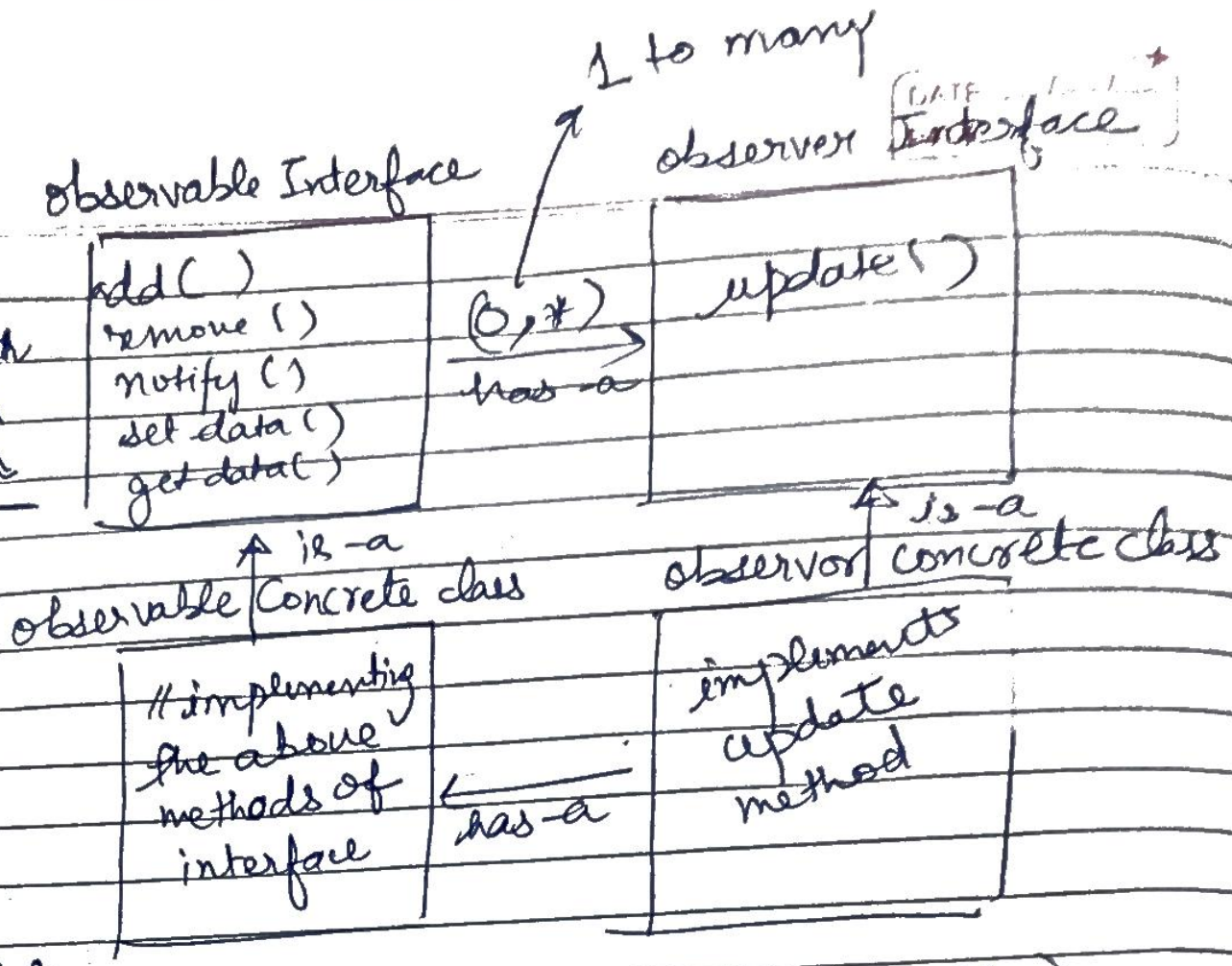
is-a

observable Concrete class

```
add(Observable Int ob) // add object to List
{ objlist.add(ob)
}
remove(Observable Int ob) → remove obj from List
{ objlist.remove(ob);
```

Scanned by Scanner Go

Flow of observer pattern Design



Real life Example:- Weather station. (observable)

curr temp.

TV Display observer

Mobile Display observer

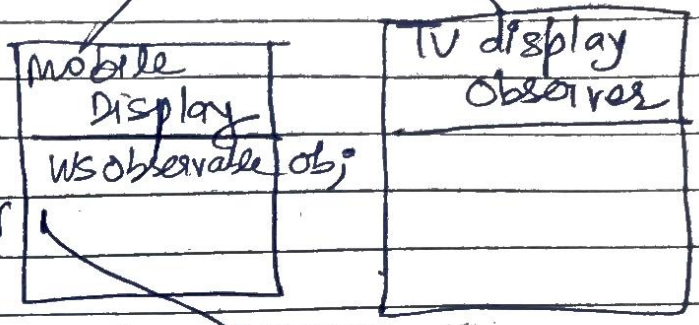
(observers)

Weather Station Interface (WS)

Scanned by Scanner Go

```
WS_observable {
List<displayobserver> objlist
  add( Displayobserver obj);
  remove( "  " );
  notify();
  set_temp();
}
```

```
Display_observer {
  update();
}
```




```
WS_observable implement {
List<displayobservable> displayList;
 int temp;

add( )
{  // implement add
}
remove( )
{  // implemnt remove
}
notify ( )
{
for( displayobserver obj : displaylist)
   {
      obj.update()
   }
}
set_data( int newTemp)
{  curr_temp = newTemp;
     notify();
}
```

Constructor call

```
MobileDisplay( wsobservable o)
{
  this.ob = o;
}
```

This is called constructor Injection

Scanned by Scanner Go

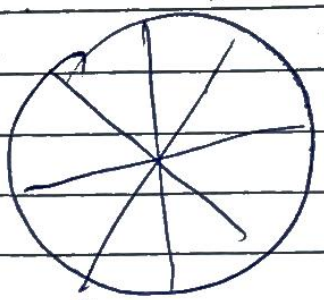
DATE :__/__/__
PAGE :

3) Decorator Pattern

The decorator Pattern is a structural pattern that allows adding new functionality to an existing object without altering its original class.

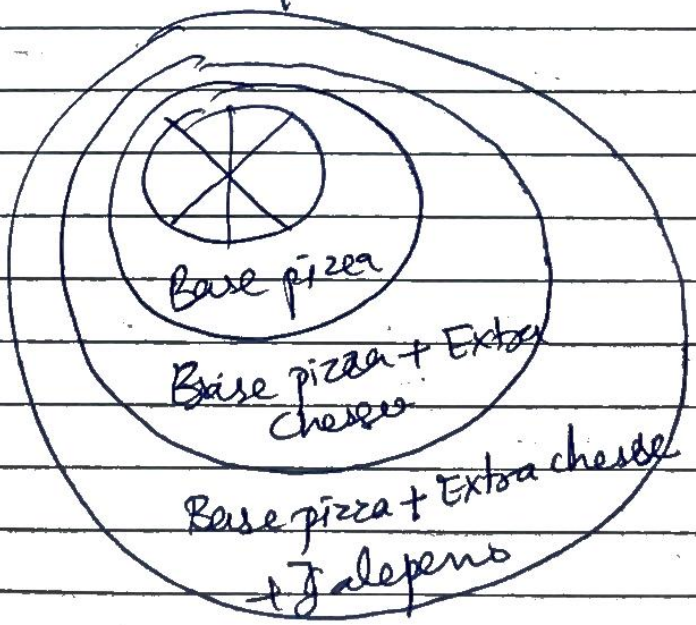
This pattern involves wrapping the original object in a decorator class, which has the same interface as the object it decorates

Use case



Base pizza.

Toppings/wrappers
- Extra cheese
- Mushroom
- Jalepeno.

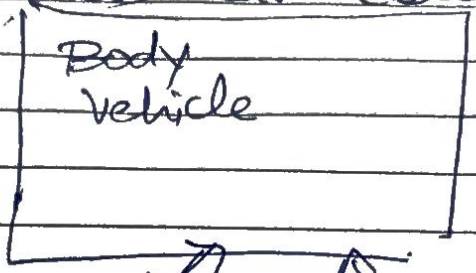


Scanned by Scanner Go

DATE : __/__/__
PAGE :

Need: To deal with class Explosion

Ex:- class car (Base)



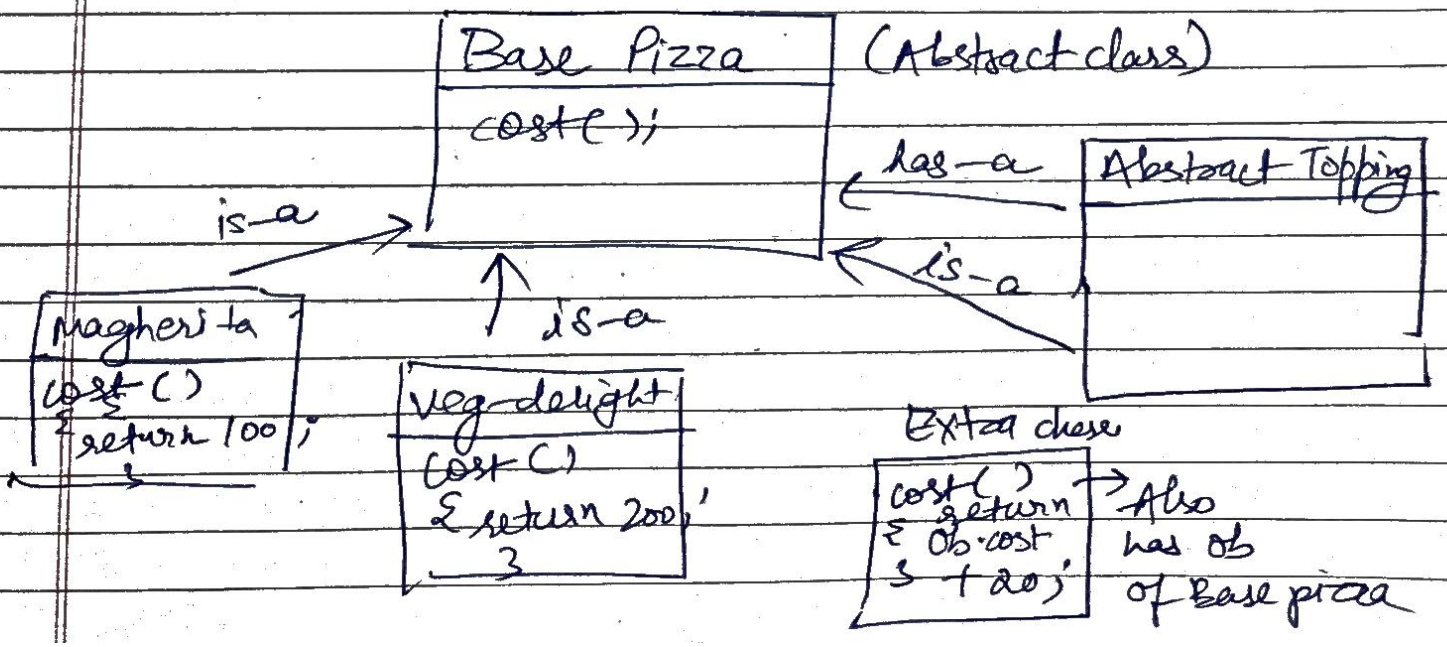
Base car + AC + Power steering + Fog light

class Base Car + AC

Base Car + AC + Power Steering

As new features are added no of classes ~~~~ deriving the orignal class is increasing.
→ Low manageability

Solution:- We can use decorator pattern in such cases to avoid class Explosion.



Scanned by Scanner Go