- Static: No diversity just business flow, ecb
- Dynamic: shows flow, behaviour, diversity. jaha baat ho, Behaviour Diagram ↓ functionalities ki

Dated:

Integrating different Modules together → component Diagram

IMPLEMENTATION DIAGRAM → Deployment Diagram

- Modules ki interaction through any operations, we are talking for .exe files. such as libraries jo hai wo component hain, we uses printf and uses this library so printf is operation.

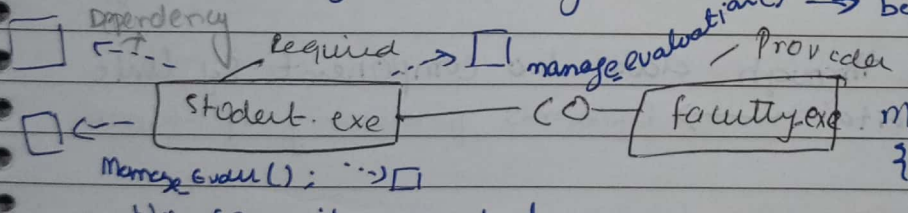
Component Diagram:

→ Compone
→ Interfac
→ Dependen
etc.

- Modules banaty hain, student hai ja usky andr multiple classes mil ky ek executable banyga (calling)
- It has required and Provider (jo
- If Student and faculty ko interact karna ho functionality ky through karengy.

Component Vs Classes

Component	Classes
• Physical things	• logical abstractions
• Higher level of abstraction	• less than component
• Only contain Operations	• both attr. and operations,



- NO compiler needed.

- Physical piece of system's implementation, including software code (source, binary or executable). It is replaceable part of sys that packages implementation and provides realization of interfaces
- It is used to model the static implementation view of system. More Abstract than classes, provides a service, executable.

DIAGRAM → Deployment Diagram

through any operations,
.exe files. such as libraries jo hai
we uses printf and uses this library
n.

→ Compone
gram: → Interfac
→ Dependen
etc.

, student hai ja
ltiple classes
utable banyga
)

and Provider (jo
ulty ko interact karna ka functionality
fy.

manage evaluation() → basically interface.

/ Provider

faculty.exe

manage_evaluation
{
}

.

lem's implementation, including software code
able). It is replaceable part of sys that

Component V/s Classes

Component	Classes
• Physical things	• logical abstractions
• Higher level of abstraction	• less than component
• Only contain operations	• both attr. and operations.

Diagram: → Components
→ Interfaces
→ Dependency, Generalization, Association etc.

train ; student hai jaisy ek module to multiple classes hongi or wo sb executable banygi, Module is executable.

implementation.

ured and Provider (jo karyga use) faculty ko interact karna to functionality

rency.

manageEvaluation() → basically interface.

Provider

faculty_exe Manage_Evaluation

}

}

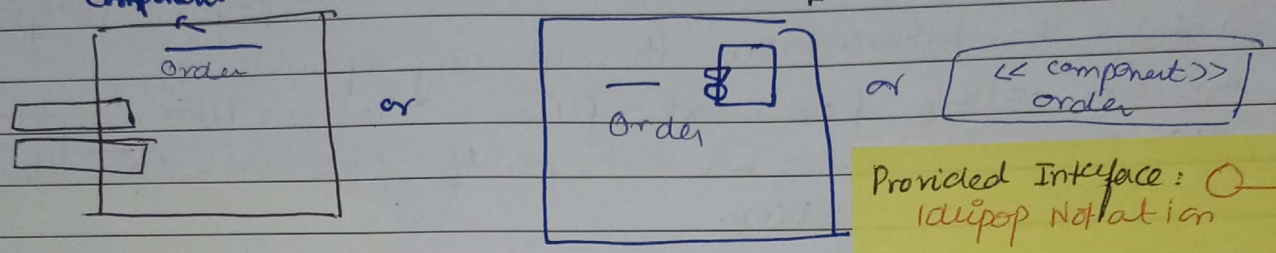
ed.

system's implementation, including software code cutable). It is replaceable part of sys that ation and provides realization of interfaces

Dated:

Notations → Deployment Component: (Dlls, Exes)
→ Work product : (source code files)

Component: 3 types of Component → Execution Comp: (com + objects)



Provided Interface: ○—
Lollipop Notation
- Characterize services that the component offers to its environment.

Required Interface:)—(
- Characterise sevice that components expect from its environment.

Solved)—

Interface → collection of one or more operations,

Provider ○— Required)—(

only operation not implementation

Dependency → Caller (required)
↓ (supplier) (provider)

Interaction Point b/w Comp and Environ

A relation its classes

□ Part allows by external sy data

y Uni ka gate is part. □

jo access karta usky sth data hai.

□ Gate 1, 2, 3, 4 (if same ↓ Aesy bandedengy working)

* Connectors:

Deligater: Part ky through data ko component sy link karna is deligator → Represents forwarding of signals.

Assembly Connecter: —(○— defined from required interface to provided service

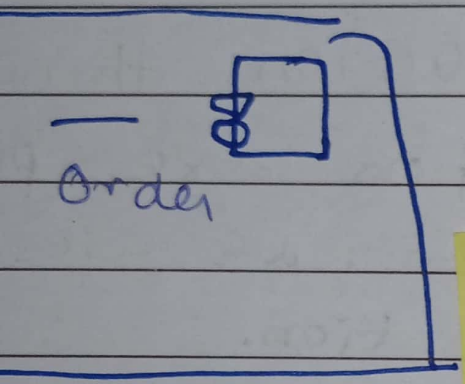
- Same function hain dono terf sy, one provides service that another component requires.

Usage dependency: One element requires another element for its full implementation. [Manager] - - - - → [Office] «uses»

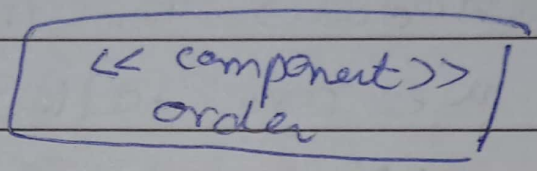
→ Deployment Component: (Dlls, Exes)

→ Work product : (source code files)

→ Execution Comp: (com + objects)



or



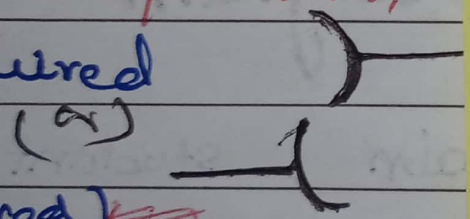
Provided Interface: ○—
Idlipop Notation

- Characterize services that the component offers to its environment.

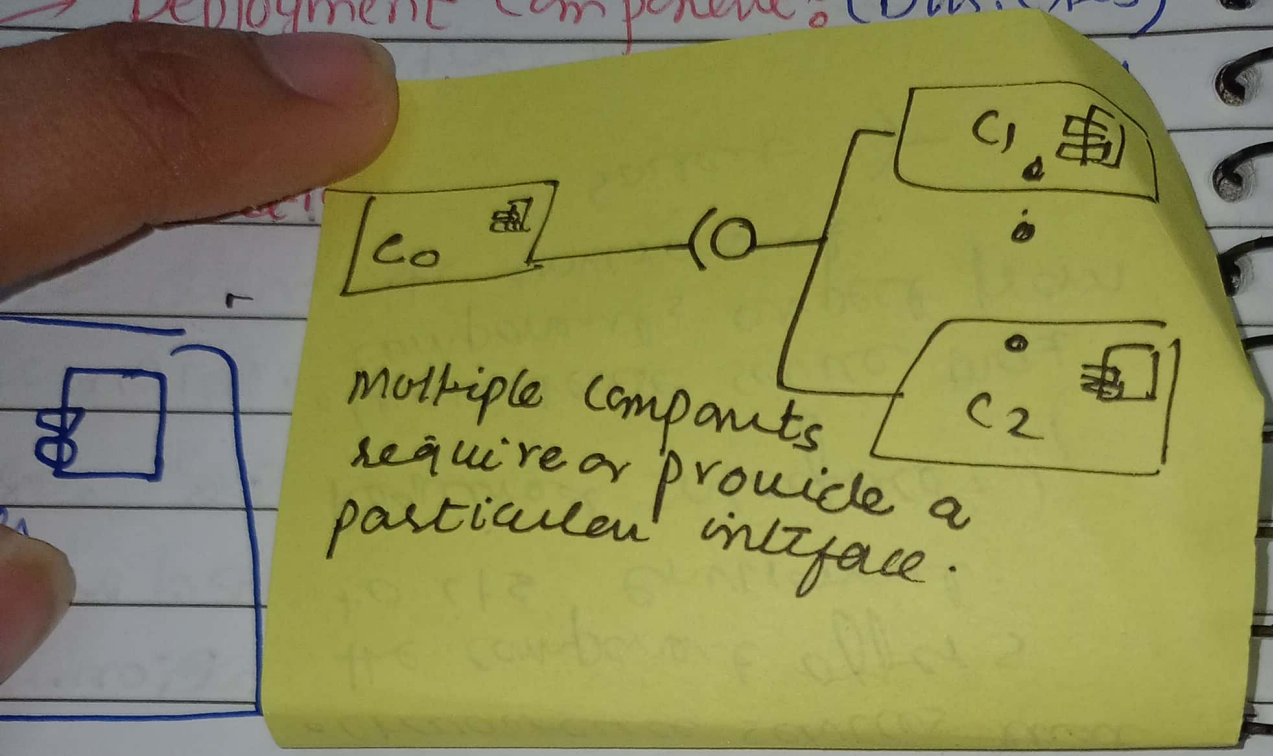
Required Interface:)—

- Characterise sevice that components expect from its environment.

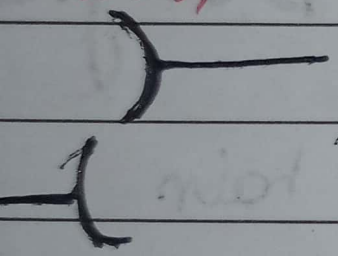
Socket)—



→ Deployment Component: (Dlls, Exes)



utions,



A relationship b/w component and
ts classes can be represented using
dater lyky agaye). dependency

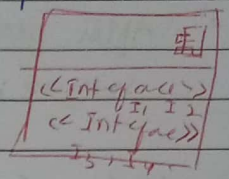
☐ Note 1, 2, 3, 4 (if same
Aesy banadengy. working)

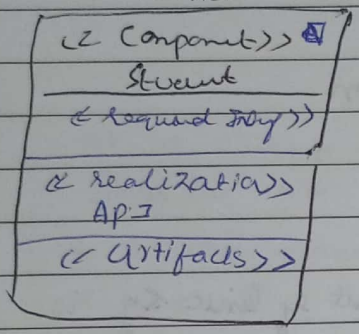
ta ka component sy link
epresents forwarding of signals.

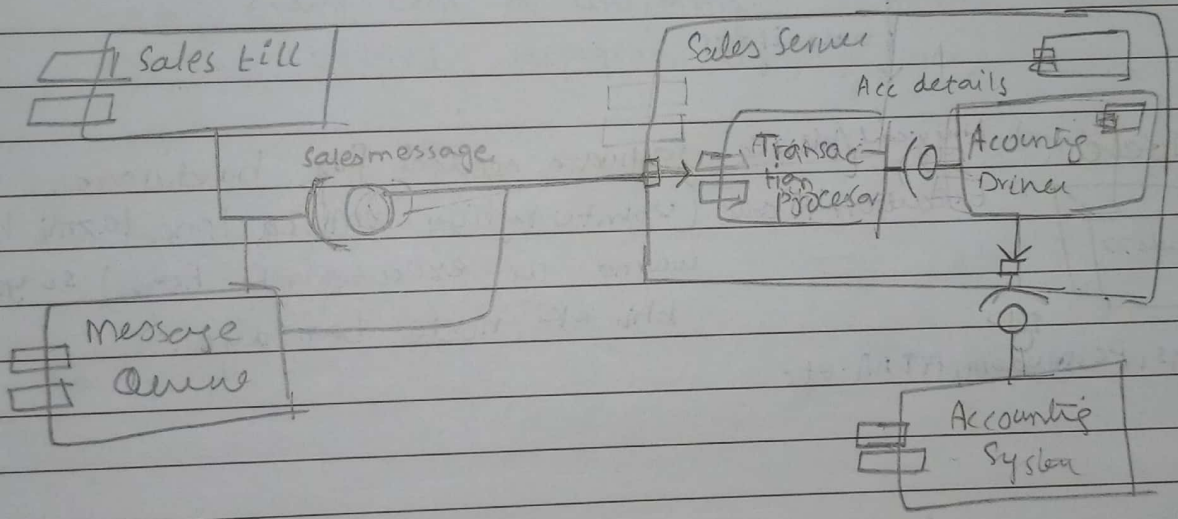
Dated:

★ Views of Component Diagram:

- External view: black box view shows publicly visible properties and operations. It is by means of "Interface symbols".
- Internal view: White box







Dated:

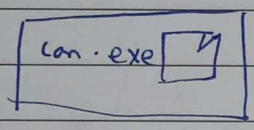
Static Diagram

DEPLOYMENT DIAGRAM

- After identifying components we are deploying on physical nodes, systems; Files → components PCs → nodes

- Nodes, components, Manifestation (realization). Association. Interface ki implementation

- Artifacts



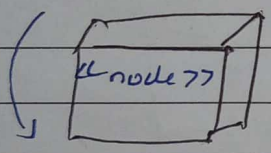
y

component ka end product. Artifacts ka component sy link sy is manifestation.

→ Manifests.

Node → Physical/device software needed for hardware.

→ Execution Node (Ubuntu kyliye VM ka hona lazmi hai warna wo execute nhi hoga) so ye bhi ek node bangaya.



Physical sys, PC, Modem, ATM etc

Timing

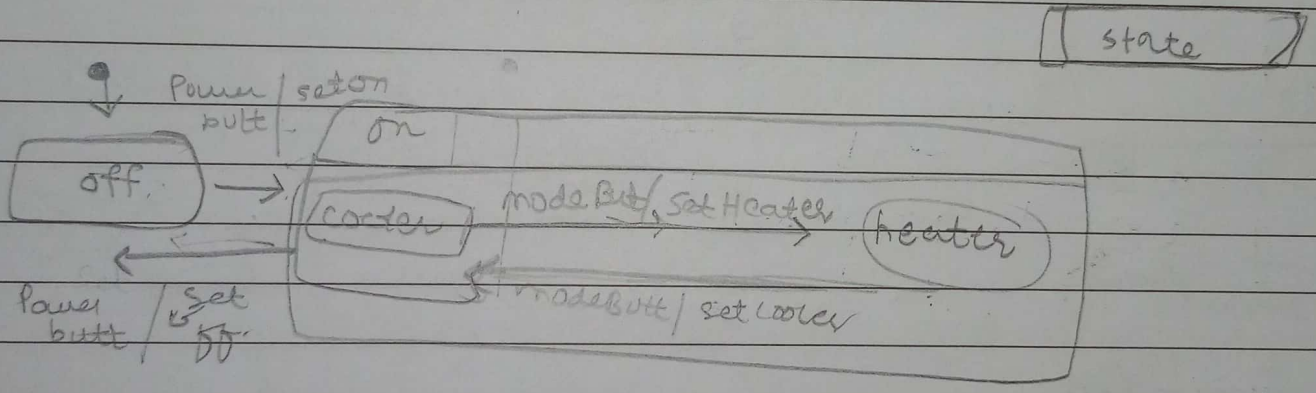
Dated:

STATE MACHINES

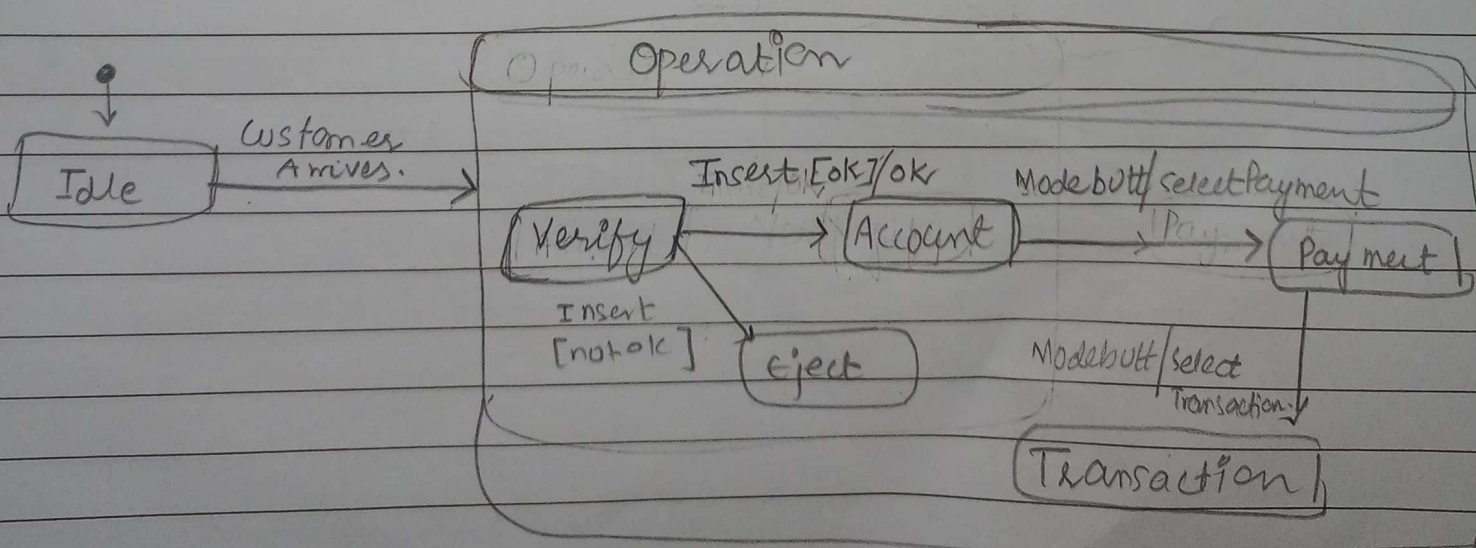
- State chart wahan banyega jahn diversity ushi ho.
- Mostly ekr hi class ki banrahi hoti hai State Chart.

→ State dependent state Independent (No diversity, No chart)

- On the basis of certain event naye action py jayega yeh. Event can be anything. Agr event nhi likhry to wo activity diagram banjayega..



Q4



General sol to a common problem in a context
what you can do — what you want — what you have

Dated:

23 design pattern exists.

Deployment Diagram

DESIGN PATTERNS (repeatable, Reusable sol)

- Behavioural Dealing with functionalities
- Structural = = Class → inheritance identifying relationships
- Creational view how objects created.

Increases sys flexibility

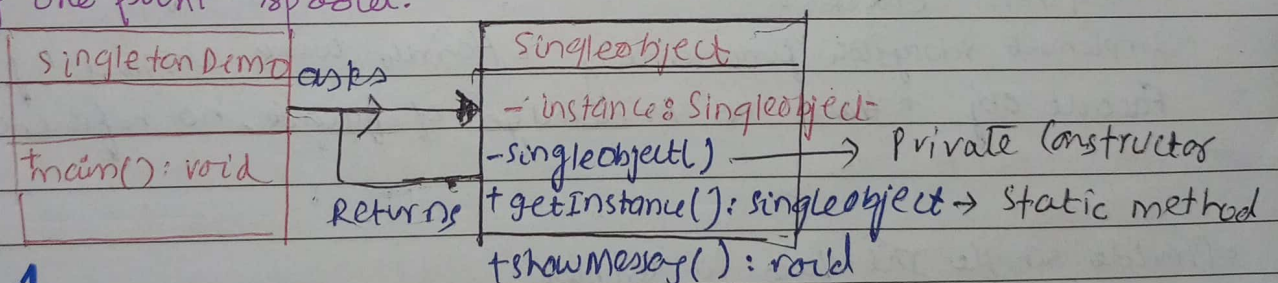
CREATIONAL PATTERN

→ single obj but linking at one place for common sol, ek hi kam par bar horha ho

★ Singleton :

- Involves single class responsible to create an object (only single object)
- Only one linking · One instance of class. And through that instance calling multiple functionalities of diff classes as self.
- Private Constructor → for restricting multiple instances.
  " Printer wali example, ky ek hi instance hai.
- Global Point Of Access to it, object can be accessed directly without need to instantiate the object of class.

Motivation: 1) Exactly one user Interface object 2) Need to create only one print spooler.



Step 1

```
public class SingleObject{
   private static SingleObject instance = new SingleObject();
   private singleObject() {}
   public static SingleObject getInstance(){
      return instance;}
   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

~~login~~ ~~register~~ part

Step 2 : Get only object from singleton class.
SingletonPatternDemo.java

```
Donate public class SingletonPatternDemo {
Blood  public static void main (String[] args)
       {
         //Error // SingleObject obj = new SingleObject();
         SingleObject object = SingleObject.getInstance();
         Object.showMessage();
       }
```

Step 3 Verify Output

Participants: 1-Static data Mem 2) private constructor 3) Static public method returns reference.
Collaborations:
Clients acess through static operation.

Applicability:
1- One instance
2- Sole instance should be extensible by subclassing, clients should be able to use the extended instance without modifying code.

Consequences:
- Controlled Access to sole instance
- Permits refinement of operations and representation
- Permits a variable no of instances.

* Single
- Involves
- Only a instan
  as self
- Privat
- Priv
- Global with

Motiva
only or

Step 1
public
priva
pri
pu

Interface ka keyword Implement.

Dated:

STRUCTURAL PATTERN → class add jo link kar rhi hogi.

FACADE PATTERN

(agy pages py code continue).

- call operator working as a facade in some restuarnt order.
- Interface ki tarh kaam kr rha hai.
- Client Facade Restaurant.
- Interface to link, and functionalities ko class mai add karwa rhy.
- Hide complexity of system, provides interface to client using which client can access system.
- functionality class k ands kr rhy thy.
- Define facade object that implements a simple interface in terms of (by delegating to) interfaces in subsystem and perform additional functionality before/after forwarding a request. Facade: knows which subsys class responsible for req.
- Delegates client requests to appropriate subsystem object.

subsystem classes:

- Implement subsystem functionality
- Handle work assigned by Facade obj
- Have no knowledge of facade, no references.

Participants

Usage:

- Provide simple Interface to complex subsystem
- When there are many dependencies b/w clients and implementation class
- You want to neatly layer your subsystem.

Dated:

Facade Shape Example:

step 1: Create Interface.

```
public interface Shape { void draw(); }
```

step 2: Create Concrete classes implementing same interface.

Rectangle.java

```
public class Rectangle Implements Shape {
  @Override
  public void draw()
{ System.out.println("Rectangle :: draw()"); }
}
```

Square.java

```
public class Square Implents Shape {
  @Override
  public void draw()
  { system.out.println("Square :: draw()"); }
}.
```

same as above 2. ← Circle.java

step 3: Create Facade Class.

```
Public class ShapeMake {
    private Shape circle; private Shape Rectangle;
      "      "     square;
    public ShapeMaker() {
        circle = new circle(); rectangle = new rectangle();
        square = new square();
    }
    public void drawC() { circle.draw(); }
      "     "    = R() { Rectyle. " (); }
      "     "    = S() { square.draw(); }
    }
```

Dated:

step 4 : Use facade to draw various type of shapes

- facadepatterndemo.java

```
public class FacadePatternDemo {
    public static void main(String [] args)
    {
        ShapeMaker shapeMaker = new ShapeMaker();
        shapeMaker.drawCircle();
        = = = S();
        = = = R();
    }
}
```

step 5 : Verify output

```
Circle :: draw()
Rectangle :: draw()
Rectangle :: draw()
```

Hotel keeper

step 1 :

```
public interface Hotel { public Menus getMenus(); }
```

step 2 :

```
public class VegRest. implements Hotel {
@Override
    public Menus getMenus()
{   VegMenu v = new VegMenu();
    return v;
}
}
```

Dated:

```
public class NonVegRes. implements Hotel { @override
    public Menus getMenus()
    {
        NonvegMenu nv= new NonVegMenu();
            return nv;
    }
}

public class both implents Hotel { @override
    public Menus getMenus()
    { bothmenu nvv = new bothmenu();
        return nvv;
    }
}
```

Step 3: public class Hotelkeeper{
public

Implement = Abstract virtual func in c++.

Dated:

. Creational:

Factory Design Pattern

- Object creation ki bacat
- In facade we were hiding functionalities in this we will be ~~creating Obj~~ hiding creation of object.
- Instance factoryclass mai banyegi
- Functionality main sy hi call hogi.
- lets a class accept instantiation to subclasses at runtime
- Refers to newly created object through a common interface
- frameworks use abstract classes to define and maintain relationships b/w objects.

Applicability:

A class can't expect class of objects it must create.
- " " wants its subclasses to specify objects it creates.
- class delegates responsibility to one of several helper sub classes.

```
Create Public class shapefactory{
public Shape getshape(string shapeType)
{ if (shapeType==null) return null;
  if ( = .equalsIgnoreCase('circle')){ return new Circle(); }
elseif( = = = ("Square")){ = = =Square(); }
 = = ( = = = ("Rect"){ = = =Rect(); }
  return null;
}} public class FactoryPattern{
  public static void main(string() args){
shapefactory shapefactory = new shapefactory;
Shape s1= shapefactory.getshape("Circle"); s1.draw();
 " s2 = " " " " ("Square"); s2.draw();
 s3 }}
```

Dated:

STRUCTURAL

ADAPTER DESIGN PATTERN