**Solid principles**

S: single responsibility

The class should be responsible for only one thing.

O: Open close principle

Extension should be preferred over modifications.

L: Liskov subscription principle

Parent class object should be able to refer to the child class object seamlessly during run time polymorphism. Example a Bird can fly let the sparrow fly but do not force the penguin to fly.

I: Interface segregation principle

Client should not be forced to use an interface if it does not need to.

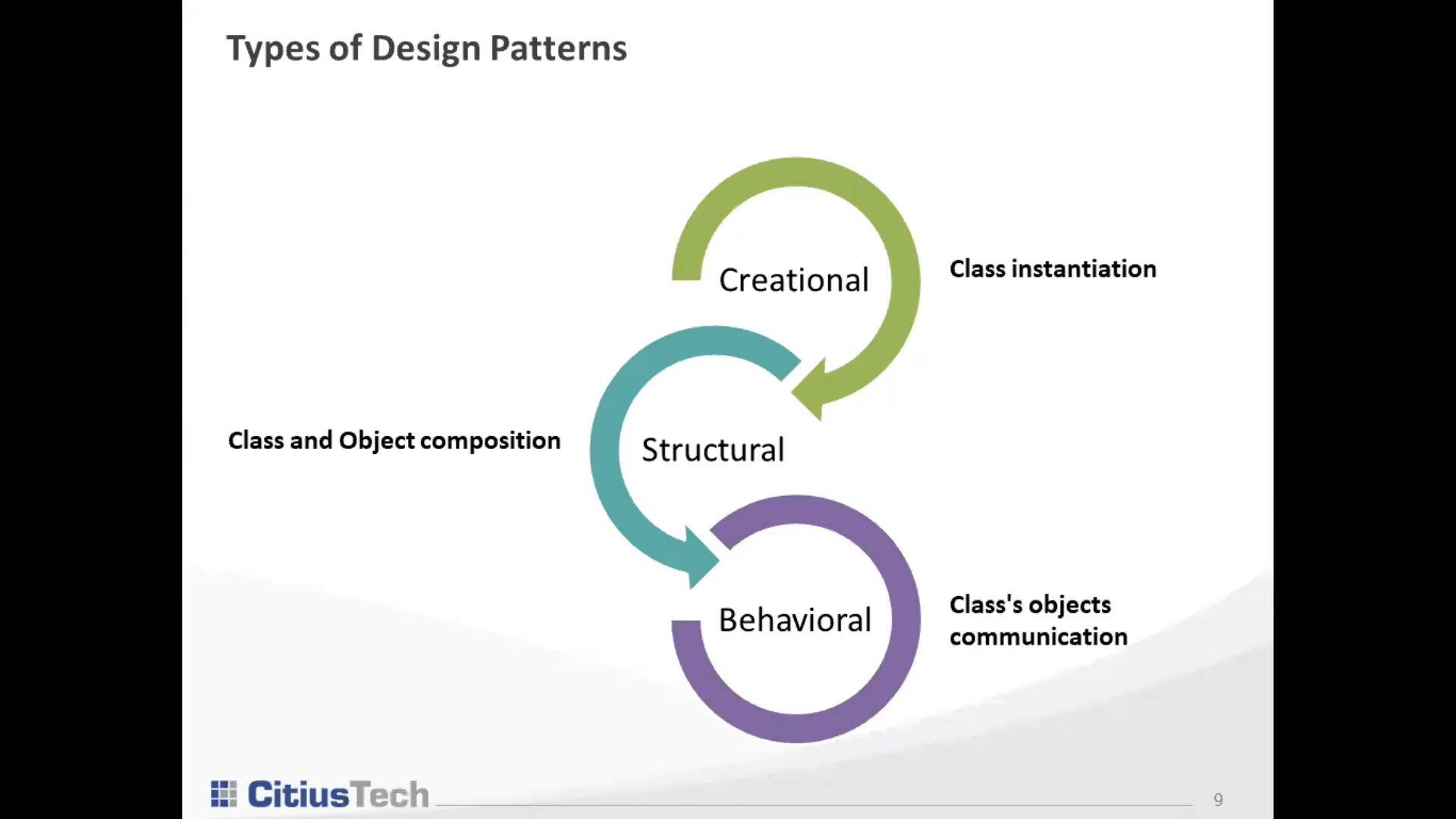
D: dependency inversion principle

High level modules should not depend on low level modules but should depend on abstraction. For more understanding refer to [dependencyInversionPrinciple](https://www.geeksforgeeks.org/dependecy-inversion-principle-solid/)

**Design patterns**

It could be description or template to solve a commonly recurring problem.

**Types of design patterns**



Creational design pattern

1 **Factory method design pattern**

Defines an interface or abstract class to create an object but lets the subclasses decide which class to instantiate. It is also called as **virtual constructor**.

So in Main method we can write

**GetPlanFactory gp = new GetPlanFactory();**

**Plan p = gp.getPlan(“Institutional”);**

As we can see Reference variable of abstract class Plan p is used to create object of class Institutional plan.

2. **Abstract Factory design pattern**

It is factory of factories. Also called as **kit**

It is one level higher than factory method design pattern

For more details refer to [Abstract factory method](https://www.youtube.com/watch?v=j50FusMmUMw).

3. **Singleton design pattern**

Only one instance of the class can be created.

**Without synchronization**

**private** **static** Singleton instance;

**public** **static** Singleton getInstance1()

{

**if** (instance == **null**) {

        instance = **new** Singleton();

    }

**return** instance;

}

**With synchronization**

Class Singleton {

private static volatile Singleton instance = null ;

// volatile keyword is used instead of synchronized as it //insure visibility in the multithreaded environment

//using synchronized keyword would be an overkill as synchronized will insure atomicity and

//visibility both

private Singleton(){}

public static Singleton getInstance(){

If(instance==null){

synchronized(Singleton.class){

If(instance==null){

Instance = new Singleton();

}

}

}

return instance;

}

}

3. **Prototype Design pattern**

Cloning of object instead of creating new one.

//create interface

Interface Prototype{

public getClone();

}

// implementing the interface

class EmployeeRecord implements Prototype{

private int id;

private String name;

private String salary;

public EmployeeRecord(int id, String name, String salary){

this.id = id;

this.name = name;

this.salary = salary;

}

@Override

public EmployeeRecord getClone(){

return new EmployeeRecord(this.id,this.name,this.salary);

}

}

public class Main{

psvm(String args[]){

EmployeeRecord e = new EmployeeRecord(1,"prashant","70000");

EmployeeRecord e2 = (EmployeeRecord)e.getClone();

}

}

4.**Builder design pattern**

It allows you to create object of a class without worrying about initialization of all the parameters of the class.

Without builder design pattern

class Phone {

private String os;

private int camera;

private String screenSize;

public Phone (String os, int camera, String screenSize){

this.os = os;

this.camera = camera;

this.screenSize = screenSize;

}

}

The only way to create the object of the above class is by intializing the constructor using all the parameters

Phone p = new Phone("android",12,"fullscreen");

//Using builder design pattern

class PhoneBuilder{

private String os;

private int camera;

private String screenSize;

public Phone setOs(String os){

this.os = os;

return this;

}

public Phone setCamera(int camera ){

this.camera = camera;

}

public Phone screenSize(String size){

this.screenSize = size;

}

public Phone getPhone(){

return new Phone(os,camera,screenSize);

}

}

///In main method we can get Phone object like this

class Main{

psvm(s a[]){

PhoneBuilder b = new PhoneBuilder();

Phone p = b.getPhone(); /// we can get Phone object without initializing any property or by partially initializing few properties.

System.out.println(p);

}

}

4.**Object pool design pattern**

Graphical user interface

Description automatically generated