Design for Change
Patterns & Principles

What's a design pattern?

- **Design patterns** are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.

- Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

Best Practices

- Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps unexperienced developers to learn software design in an easy and faster way.

History of GoF

- Over 20 years ago the iconic computer science book "Design Patterns: Elements of Reusable Object-Oriented Software" was first published.

- The four authors of the book: *Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides*, have since been dubbed "The Gang of Four".

- According to these authors design patterns are primarily based on the following principles of object orientated design.
  - Program to an interface not an implementation
  - Favor object composition over inheritance

Coding to interfaces

- Coding to interfaces is a technique to write classes based on an interface; interface that defines what the behavior of the object should be. It involves creating an interface first, defining its methods and then creating the actual class with the implementation.
  - Test Driven Development
  - Flexibility and maintainability

Favoring Composition over Inheritance

- Favoring Composition over Inheritance is a principle in object-oriented programming (OOP). Classes should achieve polymorphic behavior and code reuse by their composition rather than inheritance from a base or parent class. To get the higher design flexibility, the design principle says that composition should be favored over inheritance.

- Inheritance should only be used when subclass ‘is a’ superclass. Don’t use inheritance to get code reuse. If there is no ‘is a’ relationship, then use composition for code reuse.

- Both composition and inheritance promote code reuse through different approaches. So which one to choose? How to compare composition vs inheritance. You must have heard that in programming you should favor composition over inheritance.

Inheritance

```
class Person {
 String title;
 String name;
 int age;
}

class Employee extends Person {
 int salary;
 String title;
}
```

Composition

```
class Person {
 String title;
 String name;
 int age;

 public Person(String title, String name, String age) {
    this.title = title;
    this.name = name;
    this.age = age;
 }

}

class Employee {
 int salary;
 private Person person;

 public Employee(Person p, int salary) {
     this.person = p;
     this.salary = salary;
 }
}

Person p = new Person ("Mr.", "Kapil", 25);
Employee kapil = new Employee (p, 100000);
```

1. Inheritance is tightly coupled whereas composition is loosely coupled.

```
package com.journaldev.java.examples;

public class ClassA {

        public void foo(){
        }
}

class ClassB extends ClassA{
        public void bar(){

        }
}
```

```
package com.journaldev.java.examples;

public class ClassA {

        public void foo(){
        }

        public int bar(){
                return 0;
        }
}
```

- A new method bar() is added.
- As soon as you start using new ClassA implementation, you will get compile time error in ClassB as The return type is incompatible with ClassA.bar(). The solution would be to change either the superclass or the subclass bar() method to make them compatible.

Cont..

- If you would have used Composition over inheritance, you will never face this problem.

```
class ClassB{
        ClassA classA = new ClassA();

        public void bar(){
                classA.foo();
                classA.bar();
        }
}
```

2. There is no access control in inheritance whereas access can be restricted in composition.

```
class ClassB {

        ClassA classA = new ClassA();

        public void foo(){
                classA.foo();
        }

        public void bar(){
        }

}
```

We expose all the superclass methods to the other classes having access to subclass. So if a new method is introduced or there are security holes in the superclass, subclass becomes vulnerable. Since in composition we choose which methods to use, it's more secure than inheritance.

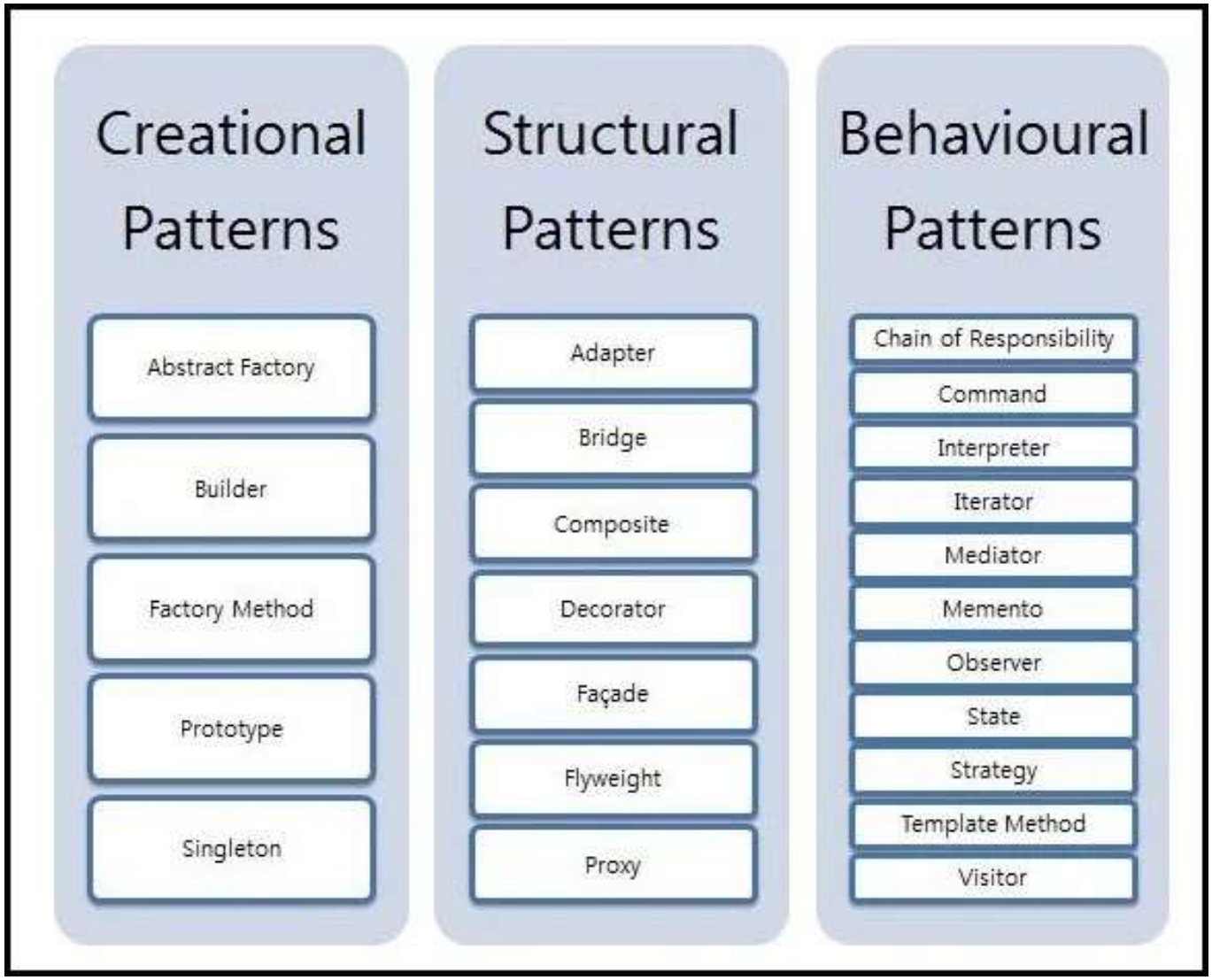
For example, we can provide ClassA foo() method exposure to other classes using below code in ClassB.

Cont..

- One more benefit of composition over inheritance is testing scope.
- Unit testing is easy in composition because we know what all methods we are using from another class.
- We can mock it up for testing whereas in inheritance we depend heavily on superclass and don’t know what all methods of superclass will be used. So we will have to test all the methods of the superclass.
- This is extra work and we need to do it unnecessarily because of inheritance.

GoF Design Pattern Types

- GoF Design Patterns are divided into three categories:
  - **Creational**: The design patterns that deal with the creation of an object.
  - **Structural**: The design patterns in this category deals with the class structure such as Inheritance and Composition.
  - **Behavioral**: This type of design patterns provide solution for the better interaction between objects, how to provide lose coupling, and flexibility to extend easily in future.


Creational Patterns
Abstract Factory
Builder
Factory Method
Prototype
Singleton
Structural Patterns
Adapter
Bridge
Composite
Decorator
Façade
Flyweight
Proxy
Behavioural Patterns
Chain of Responsibility
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template Method
Visitor

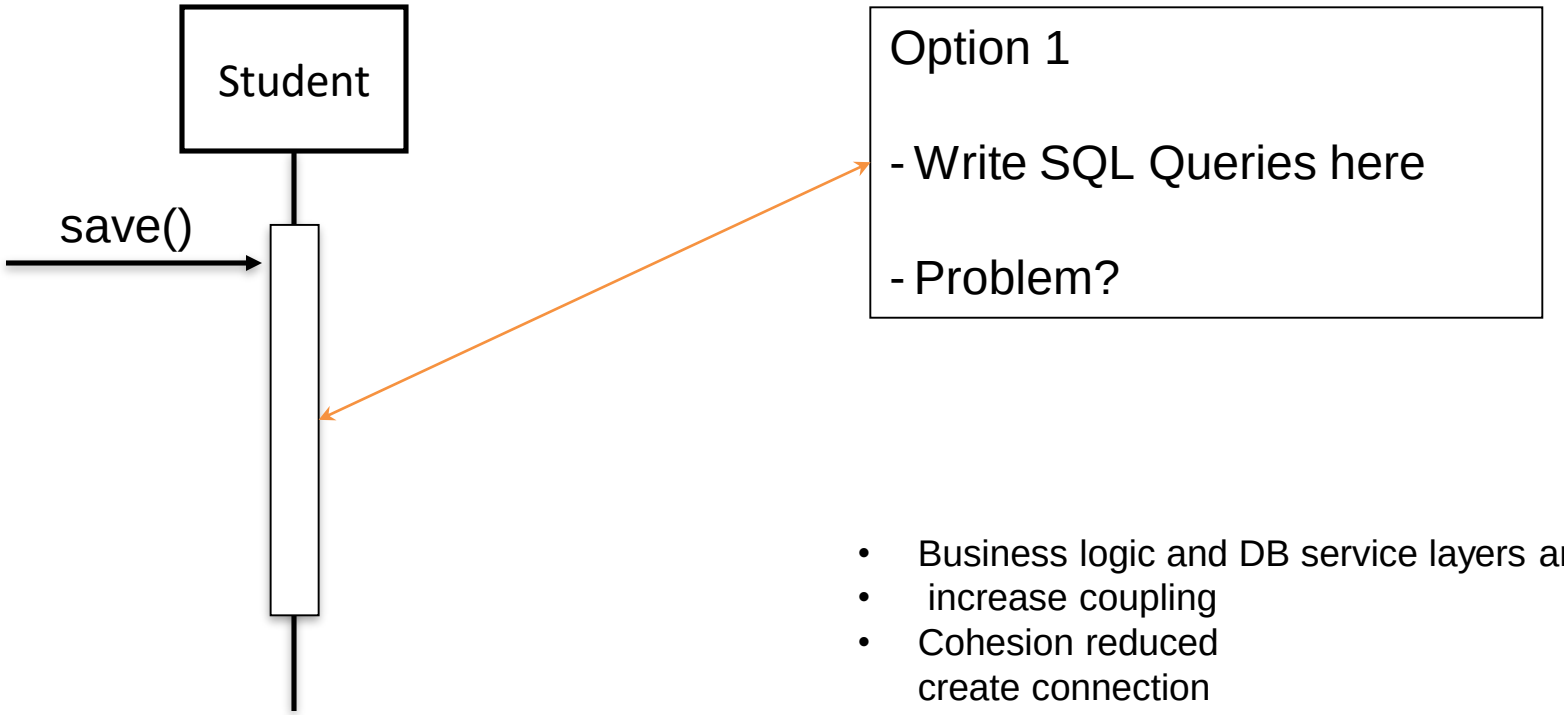
Design for Change

- Identify the functionality that may change (frequently?)
- Take special consideration in implementing such functionality in classes, so that the changing functionality will have minimized impact on the other parts
- Use the principles of Protected Variation, Polymorphism, Indirection to provide a **stable interface** of the potentially varying functionality.

Design for Change – Write to Interfaces

- The client classes are to be written in a way that they talk to the stable interface
- Example – Handling Persistence
  - Probability of change?

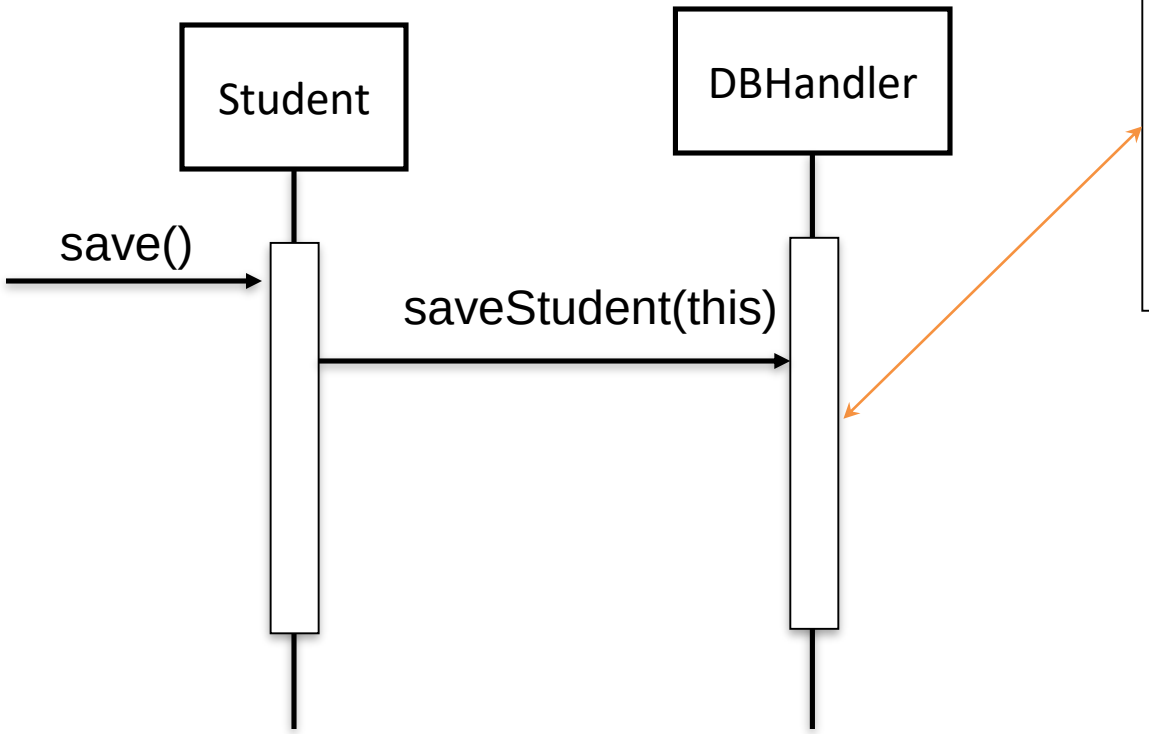
Example – Handling Persistence



- Business logic and DB service layers are merging
- increase coupling
- Cohesion reduced
  create connection
  Mention sql queries
- Impact of change is global

A good OO design can be managed through by adding new classes

Example – Handling Persistence

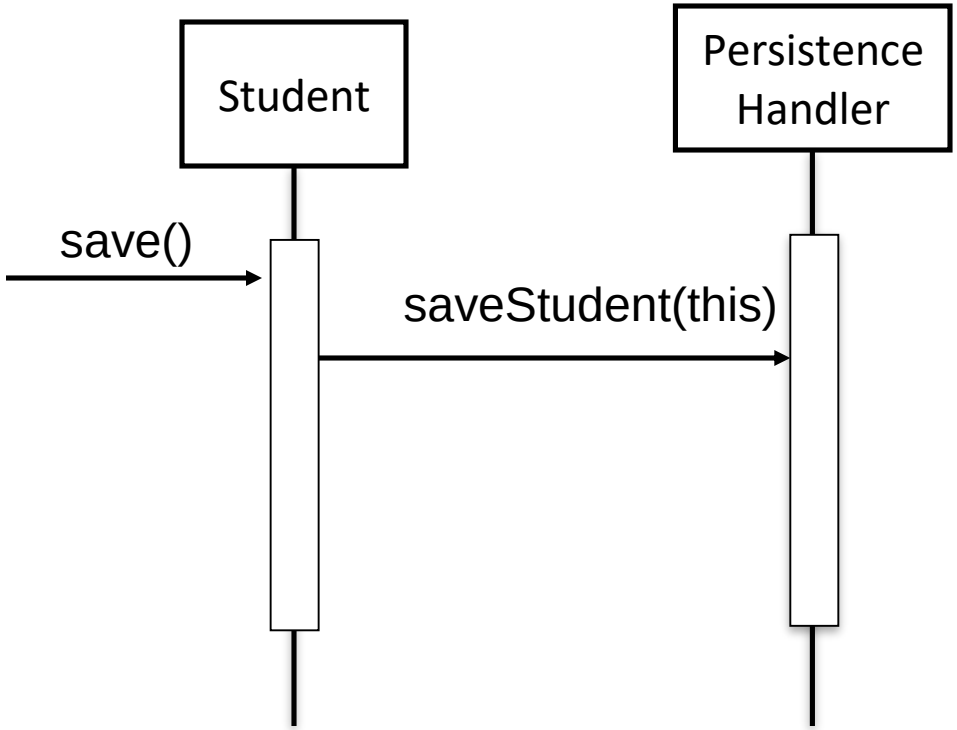


Option 2

- Write SQL Queries here

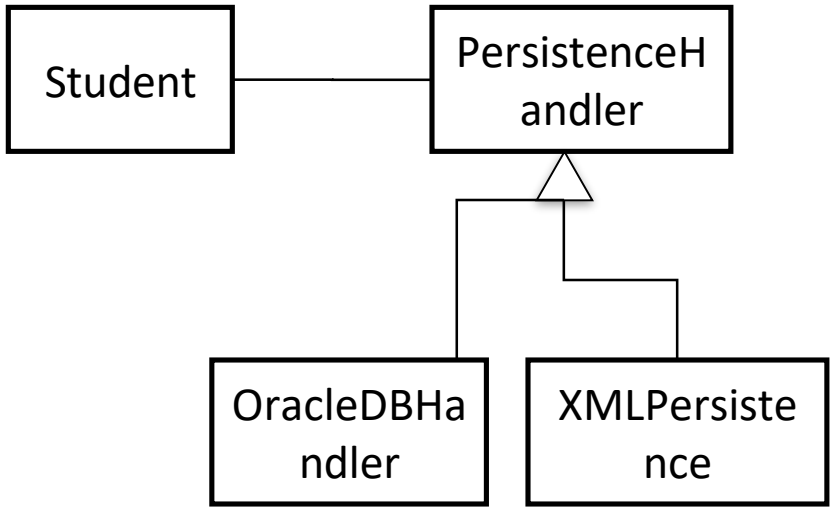
- Benefits over Option 1
- Problem?

Example – Write to Interfaces



Option 3
- Write to interfaces

- Benefits over other options
- Problem? When to initialize and how?



Implementation

```
//Student
Class Student{
                PersitenceHandler persHandler;

                 void save(){
                                persHandler.saveStudent(this);
          }

        void setPersitenceHandler (PersitenceHandler ph)
        {
                                this.persHandler=ph;
        }
```

```
// PersistenceHandler
Class PersistenceHandler{
        void saveStudent(Student s)=0;
}
```

```
class OracleDBHandler extends PersistenceHandler{

    void saveStudent(Student s){
    //connection
    //insert query formulation
    //execute query
    }
    }
```

```
class XMLHandler extends PersistenceHandler{

    void saveStudent(Student s){
    //connection
    //insert query formulation
    //execute query
    }
    }
```

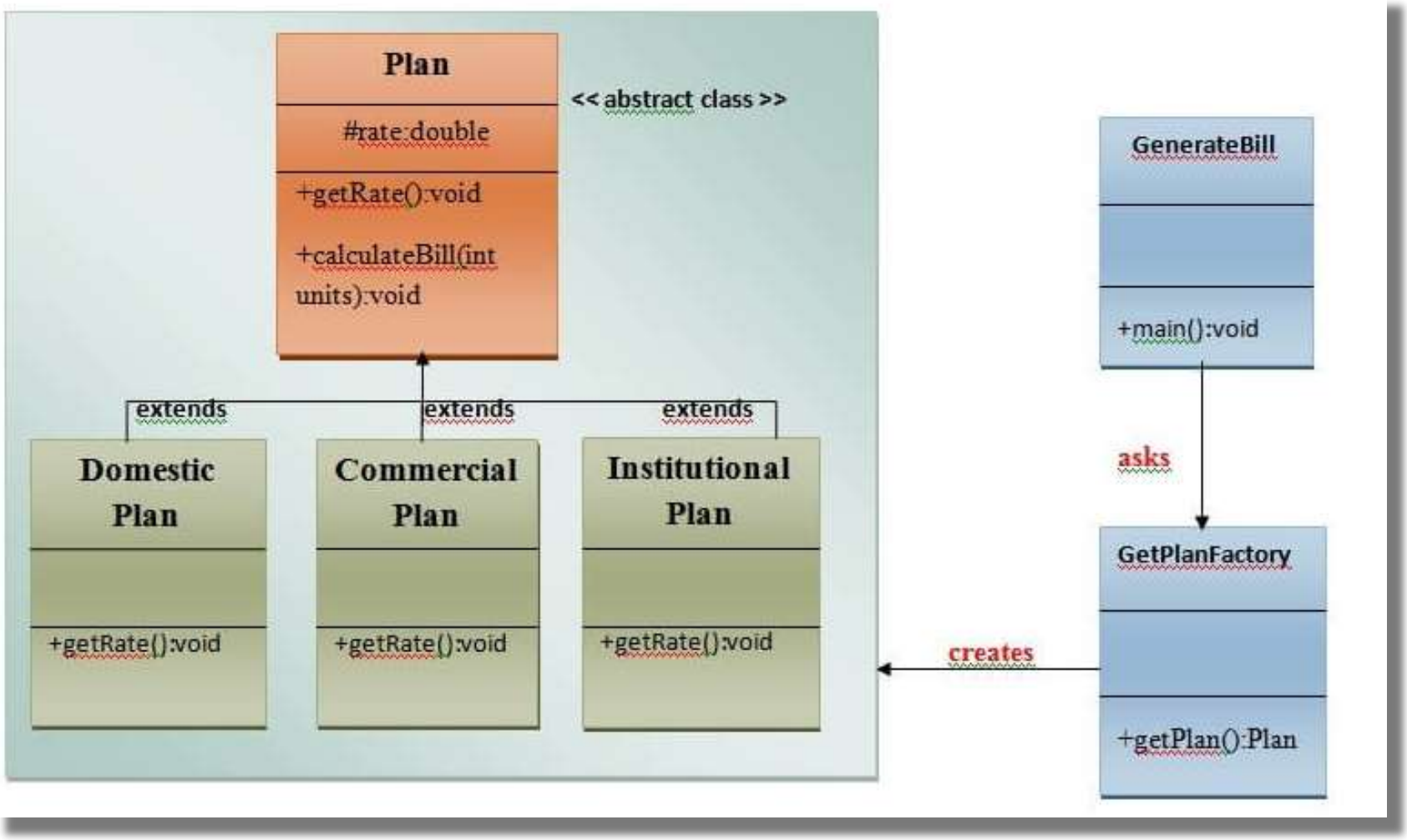
```
Void main()
{
PersitenceHandler handler= new FileHandler();

University uni= new University();

Uni. setPersitenceHandler(handler);
}
```

Factory

Creational Design Pattern

Factory

- Also called Simple Factory or Concrete Factory
- Problem
  - Who should be responsible for creating objects when there is complex creation logic or a desire to separate the creation responsibilities for better cohesion
- Solution
  - Create a Pure Fabrication object called a Factory that handles the creation


Plan
<< abstract class >>
#rate:double
+getRate():void
+calculateBill(int units):void
GenerateBill
+main():void
asks
extends
extends
extends
Domestic Plan
+getRate():void
Commercial Plan
+getRate():void
Institutional Plan
+getRate():void
creates
GetPlanFactory
+getPlan():Plan

Factory

- Who should be responsible to create PersistenceHandler class:
  - A PureFabrication object, so no information expert from domain
  - Important decision of which Persistence type to create
    - May require complex logic

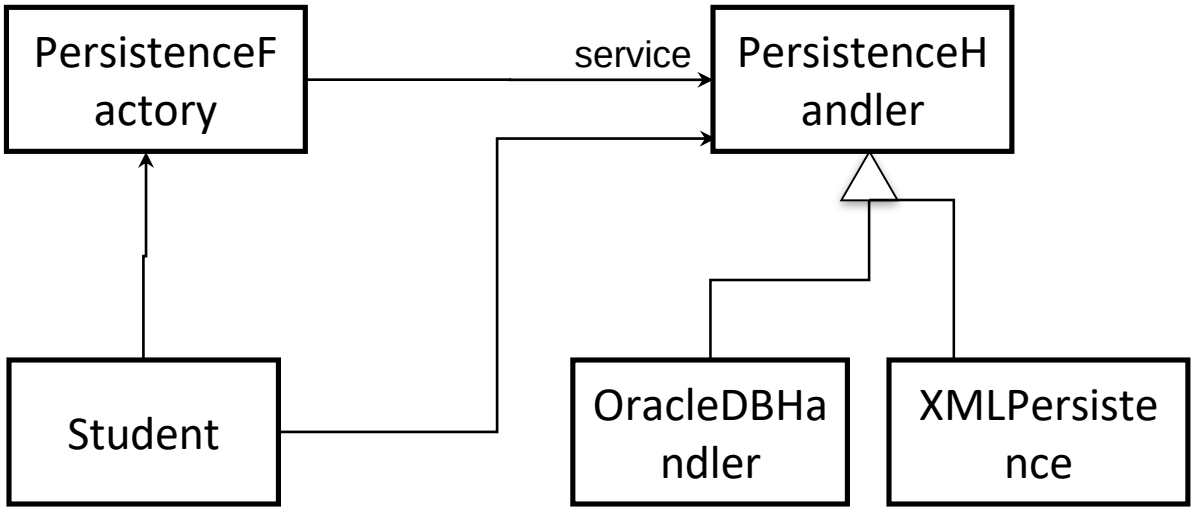
Factory

- If some domain object creates them,
  - the responsibilities of the domain object are going beyond pure application logic
- This point underscores another fundamental design principle :
  - Design to maintain a separation of concerns
- Separate distinct concerns into different areas, so that each has a cohesive purpose

Factory

- Factory objects have several advantages:
  - Separate the responsibility of complex creation into cohesive helper objects.
  - Hide potentially complex creation logic.
  - Allow introduction of performance-enhancing memory management strategies, such as object caching or recycling.

```
1  class PersistenceFactory {
2      PersistenceHandler service;
3      public PersistenceHandler createPersistenceHandler(String servName) {
4              if (service == null) {
5                if (servName.equals("Oracle")
6                    service = new OracleDBHandler();
7                else if (servName.equals("XML")
8                    service = new XMLPersistence();
9               }
10        return service;
11     }
12 }
```



Singleton

Singleton

- This pattern involves a single class which is responsible to create an object while making sure that only single object gets created.

- This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Problem

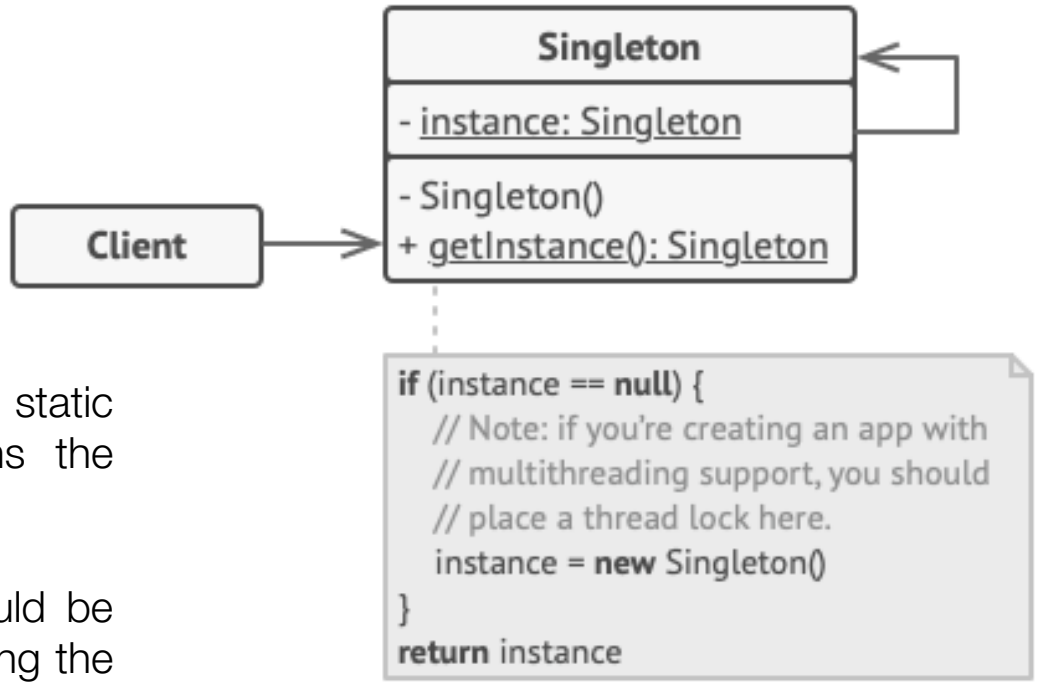
- **Ensure that a class has just a single instance.** Why would anyone want to control how many instances a class has? The most common reason for this is to control access to some shared resource—for example, a database or a file.

- **Provide a global access point to that instance**. Singleton pattern lets you access some object from anywhere in the program. However, it also protects that instance from being overwritten by other code.

Solution

- All implementations of the Singleton have these two steps in common:
  - Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.
  - Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.
- If your code has access to the Singleton class, then it's able to call the Singleton's static method. So whenever that method is called, the same object is always returned.

UML for Singleton



- The Singleton class declares the static method getInstance that returns the same instance of its own class.
- The Singleton's constructor should be hidden from the client code. Calling the getInstance method should be the only way of getting the Singleton object.

Singleton

- The PersistenceFactory raises another new problem in the design: Who creates the factory itself, and how is it accessed?
  - only one instance of the factory is needed within the process.
  - quick reflection suggests that the methods of this factory may need to be called from various places in the code
- Thus, there is a visibility problem: How to get visibility to this single PersistenceFactory instance?

```
public class PersistenceFactory {

    private static PersistenceFactory instance =null;

    private synchronized PersistenceFactory(){
    }

    public static synchronized PersistenceFactory getInstance() {
        if ( instance == null ) {
            instance = new PersistenceFactory();
        }
        return instance;
    }

}
```

```
public class Student {
     public void save() {

    PersistenceHandler dbService = PersistenceFactory.
                                   getInstance().createPersistenceHandler();

    dbService.saveStudent(this);

    }
// other methods…
} // end of class •
```