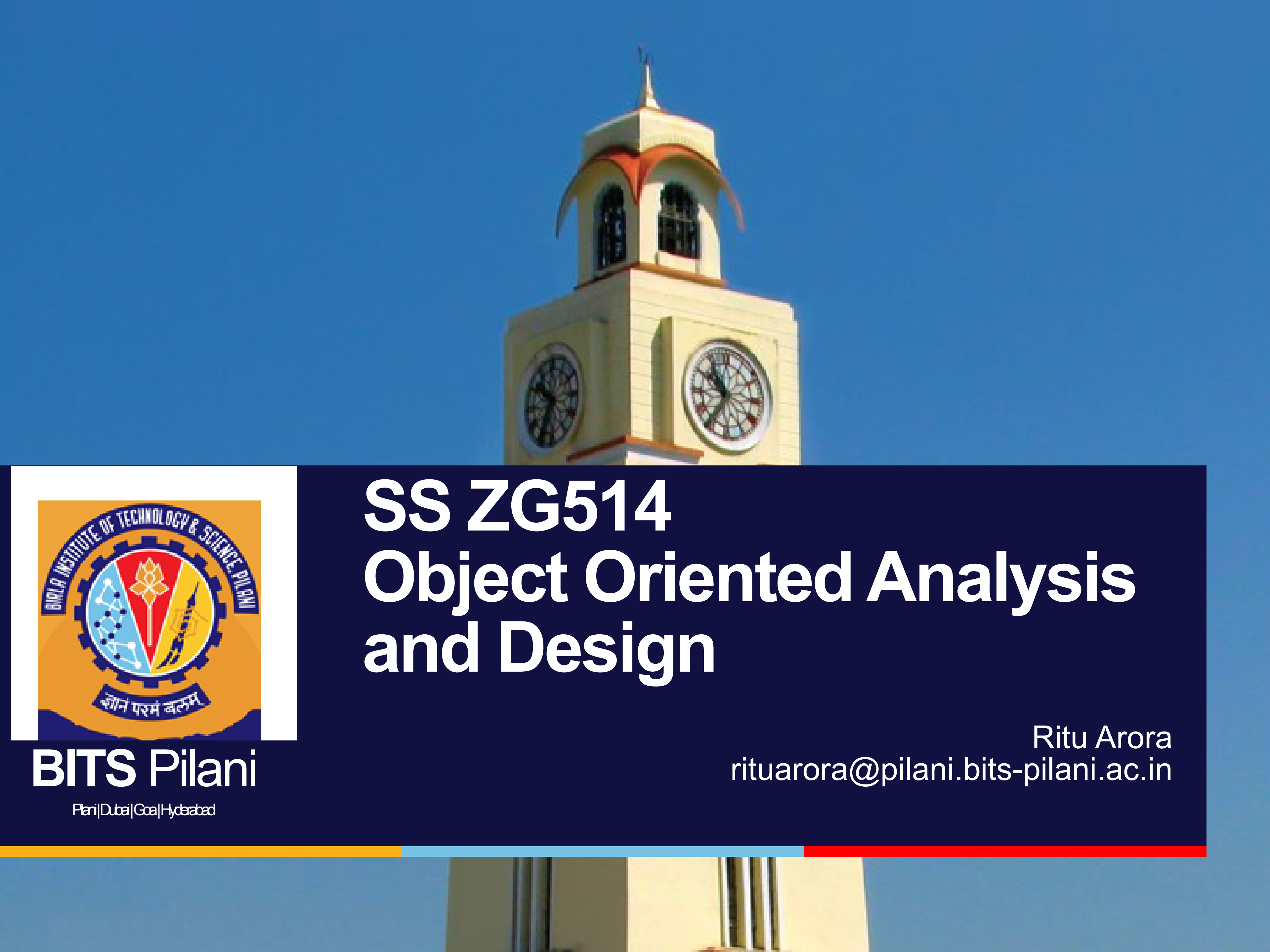


BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG514 Object Oriented Analysis and Design

Ritu Arora
rituarora@pilani.bits-pilani.ac.in

Design Patterns

Design Patterns

- In software engineering, a design pattern is a general **reusable solution** to a commonly occurring problem in software design.

- A design pattern is a **description or template** for how to solve a problem that can be used in many different situations.

- Object-oriented design patterns typically show **relationships and interactions between classes** or objects, without specifying the final application classes or objects that are involved.

Benefits of Design Patterns

- Design patterns encourage code reuse and accommodate change.
- Design patterns can speed up the development process by providing tested, proven development paradigms.
- Design patterns encourage more legible and maintainable code by following well-understood paths.

GOF Design Patterns

- Gang-Of-Four (GOF) Design Patterns were proposed by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides in the year 1994.

- There are 23 design patterns that have been categorized into 3 categories:
    - Creational patterns
    - Structural design patterns
    - Behavioural design patterns

Type of Design Patterns

- **Creational patterns** : patterns provide instantiation mechanisms, making it easier to create objects in a way that suits the situation.
- **Structural design patterns** : generally deal with relationships between entities, making it easier for these entities to work together.
- **Behavioural design patterns** : patterns are used in communication between entities and make it easier and more flexible for these entities to communicate.

Creational Patterns

- These design patterns are concerned about class instantiation.
- It aims to make object creation easier so that clients will not contain large, complex code to instantiate an object.
- Creational patterns are ones that create objects for you, rather than having you instantiate objects directly.

Design Patterns

Creational Patterns

- Factory Method
- Singleton
- Builder
- Abstract Factory
- Prototype

Factory Pattern

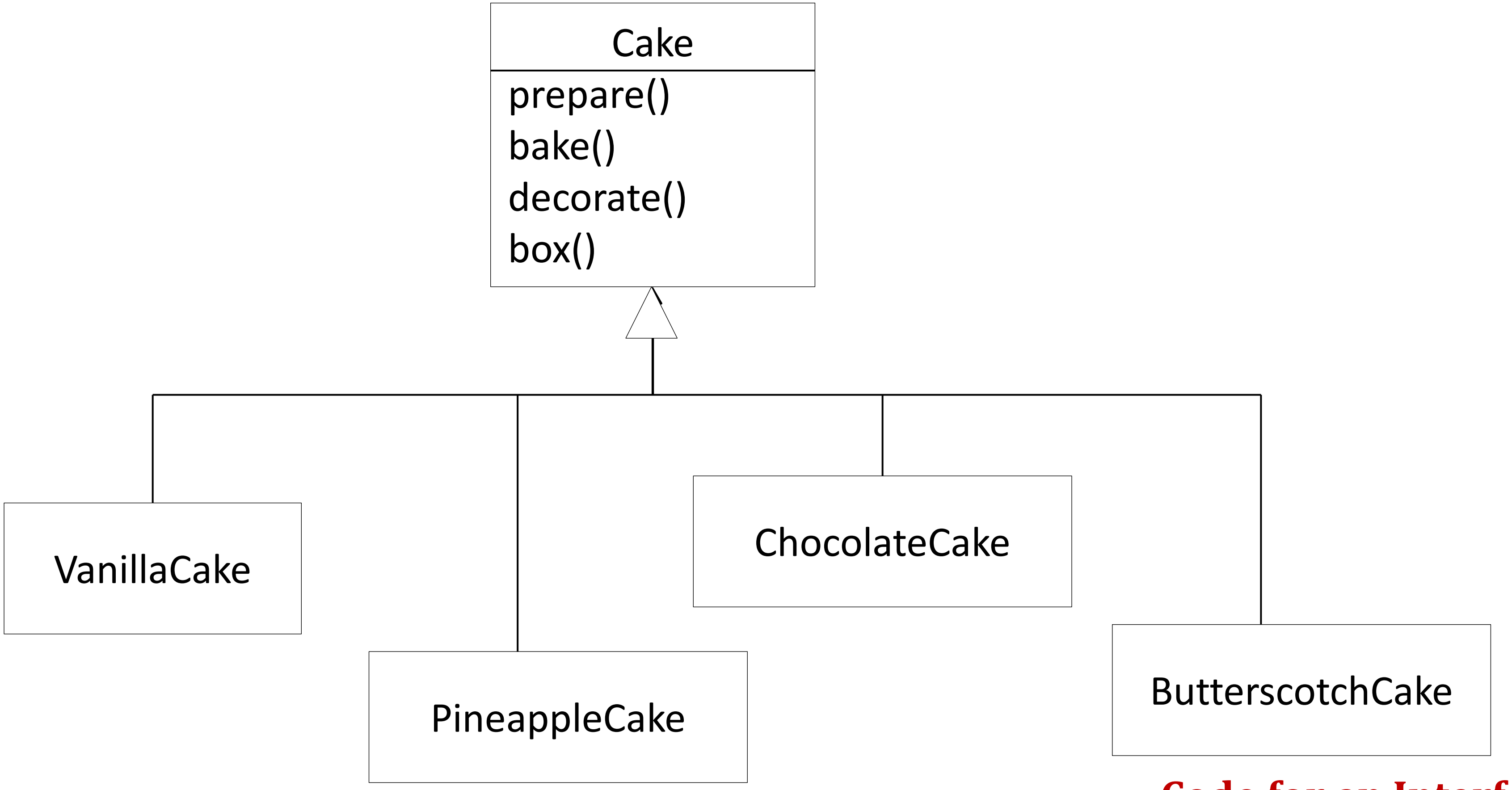
Name: Factory

Problem: Who should be responsible for creating objects when there are special considerations, such as complex creation logic, a desire to separate the creation responsibilities for better cohesion.

Solution: Create an object called a Factory that handles the creation.

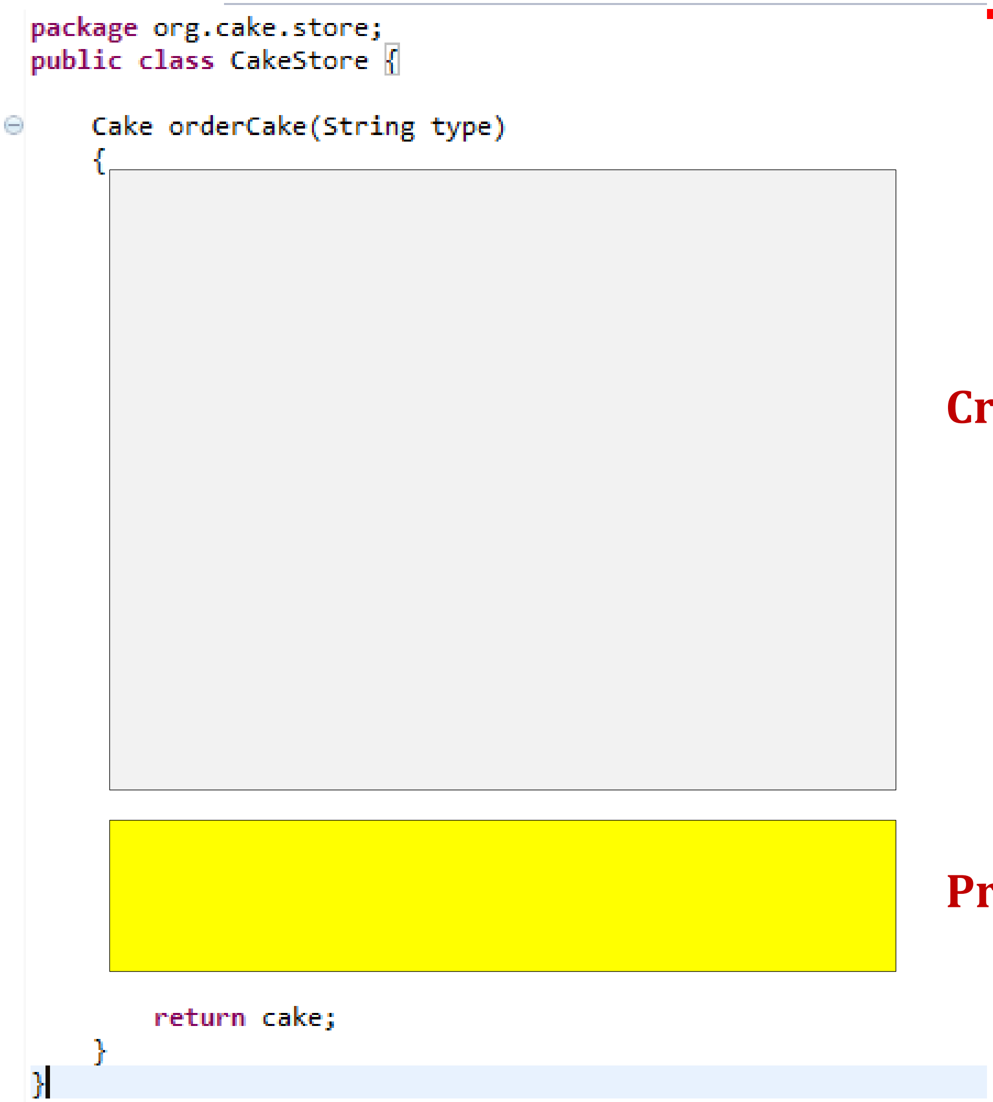
Factory Pattern

Example: Suppose that a cake store offers four type of cakes. It has the following design and code:



Code for an Interface

Factory Pattern



```
package org.cake.store;
public class CakeStore {

    Cake orderCake(String type)
    {




        return cake;
    }
}
```

Creation

Preparation

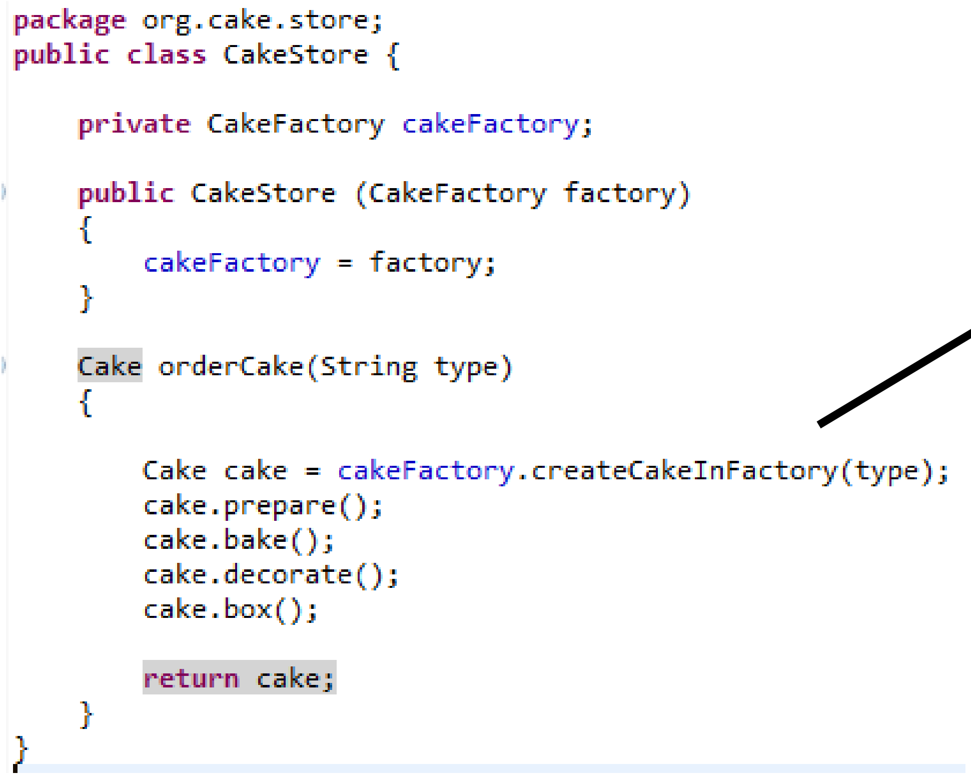
Factory Pattern

- The CakeStore class consists of code for two processes: creation and preparation of the cakes.
- This violates the cohesion principle.

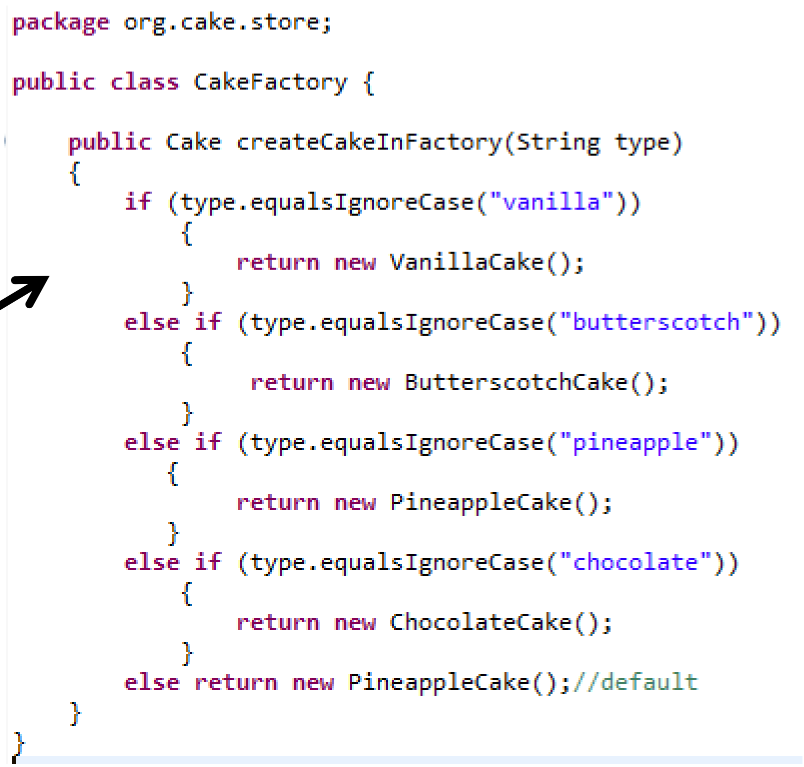
- A better way is to move the responsibility of creation of objects into another class.

- This class is termed as a Factory class.

Factory Pattern



```java
package org.cake.store;
public class CakeStore {

    private CakeFactory cakeFactory;

    public CakeStore (CakeFactory factory)
    {
        cakeFactory = factory;
    }

    Cake orderCake(String type)
    {

        Cake cake = cakeFactory.createCakeInFactory(type);
        cake.prepare();
        cake.bake();
        cake.decorate();
        cake.box();

        return cake;
    }
}
```



```java
package org.cake.store;

public class CakeFactory {

    public Cake createCakeInFactory(String type)
    {
        if (type.equalsIgnoreCase("vanilla"))
            {
                return new VanillaCake();
            }
        else if (type.equalsIgnoreCase("butterscotch"))
            {
                 return new ButterscotchCake();
            }
        else if (type.equalsIgnoreCase("pineapple"))
           {
                return new PineappleCake();
           }
        else if (type.equalsIgnoreCase("chocolate"))
            {
                return new ChocolateCake();
            }
        else return new PineappleCake();//default
    }
}
```

Factory Pattern

- Factories handle the detail of object creation

- The Factory Method Design Pattern allows us to:
  - place abstract, “code to an interface” code in a superclass
  - place object creation code in a subclass

Singleton Pattern

- The Singleton pattern ensures that a class is only instantiated once and provides a global access point for this instance.

- Examples:
  - A single instance of DatabaseManager for managing access to database.
  - A single instance of the ErrorLogManager.
  - There should be only one instance of a WindowManager.
  - There should be only one instance of a FileSystem.
  - There should be only one instance of a ServiceFactory.

Singleton Pattern

- How do we ensure that a class has only one instance and that the instance is easily accessible?

- A global variable makes an object accessible,
    - but does not prevent creation of multiple objects.
    - violates encapsulation.

- A better solution is to make the class itself responsible for keeping track of its sole instance.

- The class ensures that no other instance can be created and it provides a way to access the instance.

Singleton Pattern

Name: Singleton Pattern

Problem: Exactly one instance of a class is allowed.

Objects need a global and single point of access.

Context: In some applications it is important to have exactly one instance of a class.

Forces: Can make an object globally accessible as a global variable, but this violates encapsulation.

Could use class (static) operations and attributes, but polymorphic redefinition is not always possible.

Singleton Pattern

Solution:

- Define a static method of the class that returns the singleton.
- Create a class with a class operation `getInstance()`.
- When class is first accessed, this creates relevant object instance and returns this object to the client.
- On subsequent calls of `getInstance()`, no new instance is created, but instance of existing object is returned.

Singleton Pattern

```
public class ServiceFactory {

private static ServiceFactory instance = new ServiceFactory();

private ServiceFactory() {};

public static ServiceFactory getInstance()
{
   return instance;
}

// other methods

}
```

Eager initialization

Singleton Pattern

```
public class ServiceFactory {

private static ServiceFactory instance = null;

private ServiceFactory() {};

public static synchronized ServiceFactory getInstance()
{
    if ( instance == null) {

    //critical section if multithreaded application
    instance = new ServiceFactoty();
    }
    return instance;
}

// other methods

}
```

Lazy initialization

Singleton Pattern

Lazy initialization vs Eager initialization

- In multi-threaded applications, the creation step of the lazy initialization logic is a critical section requiring thread concurrency control.

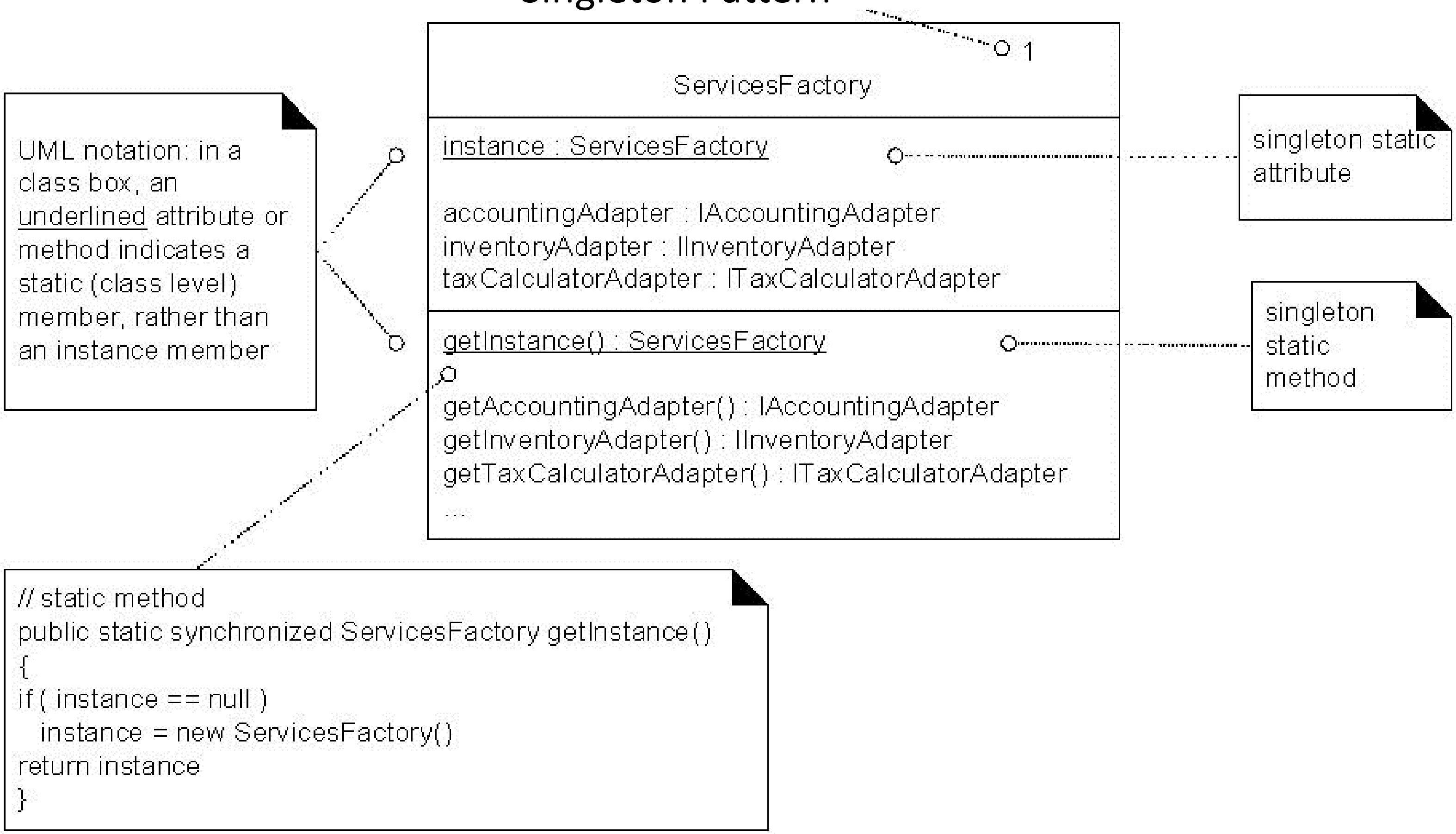
- Lazy initialization is usually preferred:
  - Creation work (and perhaps holding on to expensive resources) is avoided, if the instance is never actually accessed.
  - The **`getInstance()`** lazy initialization sometimes contains complex and conditional creation logic.

Singleton Pattern

```
public class InitializeSystem{

public void initialize()
{
   //do some work
   handleToRequiredObject=
   ServiceFactory.getInstance().getSomething();
   }
    //do some work
}

// other methods


}
```

Singleton Pattern: UML notations

Singleton Pattern



1

ServicesFactory

instance : ServicesFactory

accountingAdapter : IAccountingAdapter
inventoryAdapter : IInventoryAdapter
taxCalculatorAdapter : ITaxCalculatorAdapter

getInstance() : ServicesFactory

getAccountingAdapter() : IAccountingAdapter
getInventoryAdapter() : IInventoryAdapter
getTaxCalculatorAdapter() : ITaxCalculatorAdapter
...

UML notation: in a class box, an underlined attribute or method indicates a static (class level) member, rather than an instance member

singleton static attribute

singleton static method

```
// static method
public static synchronized ServicesFactory getInstance()
{
if ( instance == null )
  instance = new ServicesFactory()
return instance
}
```

Courtesy: Adapted from Applying UML and Patterns, Craig Larman, 3rd edition

Exercise

- Implement the CakeStore example, using Singleton Pattern for the CakeFactory.

Solution: CakeFactory

```
package org.cake.store;
public class CakeFactory {
private static CakeFactory instance = new CakeFactory();
private CakeFactory() {};
public static CakeFactory getInstance()
{    return instance;
}

public Cake createCakeInFactory(String type) {
if (type.equalsIgnoreCase("vanilla"))
{    return new VanillaCake();
}
else if (type.equalsIgnoreCase("butterscotch"))
{    return new ButterscotchCake();
}
else if (type.equalsIgnoreCase("pineapple"))
{  return new PineappleCake();
}
else if (type.equalsIgnoreCase("chocolate"))
{    return new ChocolateCake();
}
else return new PineappleCake();//default
}
}
```

Solution: CakeStore

```
package org.cake.store;
public class CakeStore {

    private CakeFactory cakeFactory;
    public CakeStore ()
    {    cakeFactory = cakeFactory.getInstance();
    }

    Cake orderCake(String type)
    {
         Cake cake = cakeFactory.createCakeInFactory(type);
         cake.prepare();
         cake.bake();
         cake.decorate();
         cake.box();
         return cake;
    }
}
```

Code to an Interface

- Each time we invoke the “new” command to create a new object, we violate the “Code to an Interface” design principle.

Example

- A computer monitor is designed for display purposes. So, the computer is a product and the computer monitor is a part or module of the computer which is responsible for display operation.

Code to an Interface

```
public class Computer
{
    public void display(){
      System.out.println(“Display through Monitor”);
    }

    public static void main(String args[]){
      Computer cm =new Computer();
      this.display();
   }
}
```

Now, there is a need to change the display on to a projector.

Code to an Interface

```
public class Computer
{
    public void displayMonitor(){
      System.out.println(“Display through Monitor”);
    }

    public void displayProjector(){
      System.out.println(“Display through Projector”);
    }

    public static void main(String args[]){
      Computer cm =new Computer();
      if (args[0].equals("Monitor"))
         {
                 this.displayMonitor();
         }
         else this.displayProjector();
   }
}
```

Code to an Interface

- Now, there is a need to change the display on to another device!!

- This is not a good design.

- As per the open-closed principle also, classes should be open for extension and closed for modifications.
- However, in this case, they aren’t open for extension.
- Additionally, we are all the time modifying the classes, which is incorrect.

- Let’s take a look at the following code:

Code to an Interface

```
interface displayModule
{
   public void display();
}

public class Monitor implements displayModule
{
   public void display(){
    System.out.println(“Display through Monitor”);
    }
}

public class Projector implements displayModule
{
    public void display(){
    System.out.println(“Display through projector”);
    }
}
```

Code to an Interface

```
public class Computer
{
    displayModule dm=null;// programming through interface
    public void setDisplayModule(displayModule dm){
    this.dm=dm;
    }
    public void display(){
    dm.display();
    }
    public static void main(String args[]){
    Computer cm =new Computer();
     if (args[0].equals("Monitor")){
               dm = new Monitor();  }
        else   dm = new Projector();
        cm.display();
   }
}
```

Code to an Interface

- So we see here that we have created an interface called **`displayModule`**, and all display equipment must implement that interface and provide its own implementation of the display operation.

Structural Design Patterns

Structural Design Patterns

- Deal with relationships between entities, making it easier for these entities to work together.
- These design patterns are all about Class and Object composition.
- How to compose and relate classes and objects to form larger structures.

Structural Patterns

- Adapter
- Composite
- Facade
- Bridge
- Decorator
- Flyweight
- Proxy

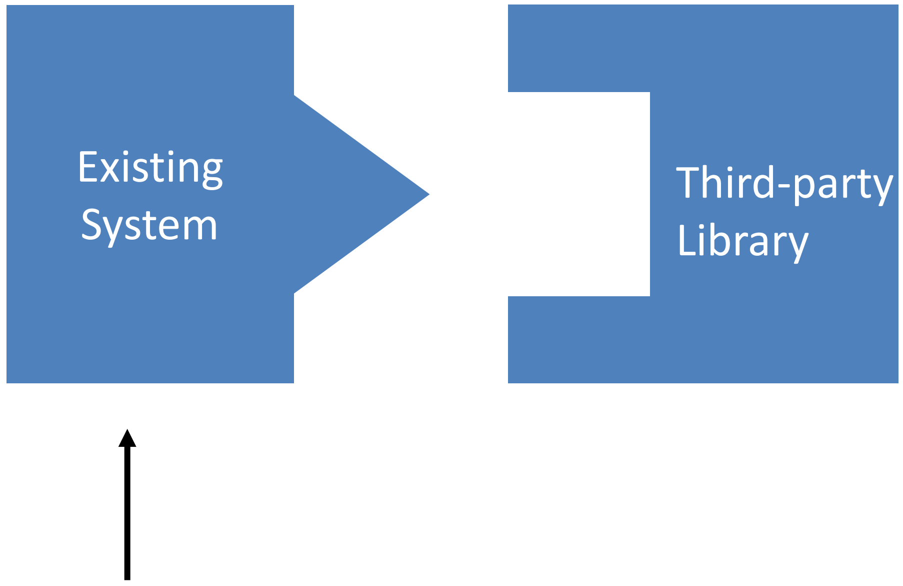
Adapter Pattern

Name: Adapter

Problem: How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?

Solution: Convert the original interface of a component into another interface, through an intermediate adapter object.

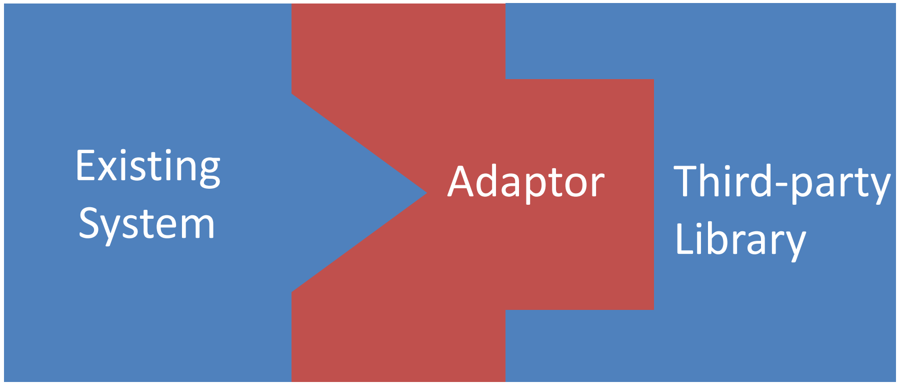
Adapter: What do they look like??



Being accessed by the client

Interface mismatch: Need an Adapter

Adapter: What do they look like??



Being accessed by the client

Client can continue to use the existing system, but the third-party library gets added

Adapter Pattern

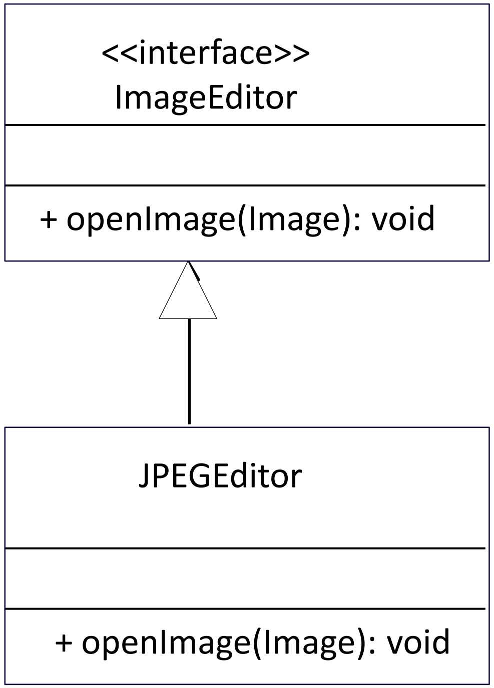
- Convert the interface of a class into another interface that the client expects.
- Adapter lets classes work together that could not otherwise do so, because of incompatible interfaces.
- A common example of Adapter (outside software domain) is the use of a two-pin converter

Adapter Pattern: Example

- Consider that Company ABC has a image editing software that can open and edit images of JPEG format.
- This interface is already in use by the existing clients.
- Now, suppose one of the client wants that they should be able to open/edit images of TIFF and PNG format as well.
- A third-party library exists which provides the required facility.
- Now, it is the responsibility of Company ABC, to add the newly required facility, without hampering the existing interface, since there are other clients who are using the existing interface.

Adapter Pattern

- Details of the image editing software that can open and edit images only of JPEG format.

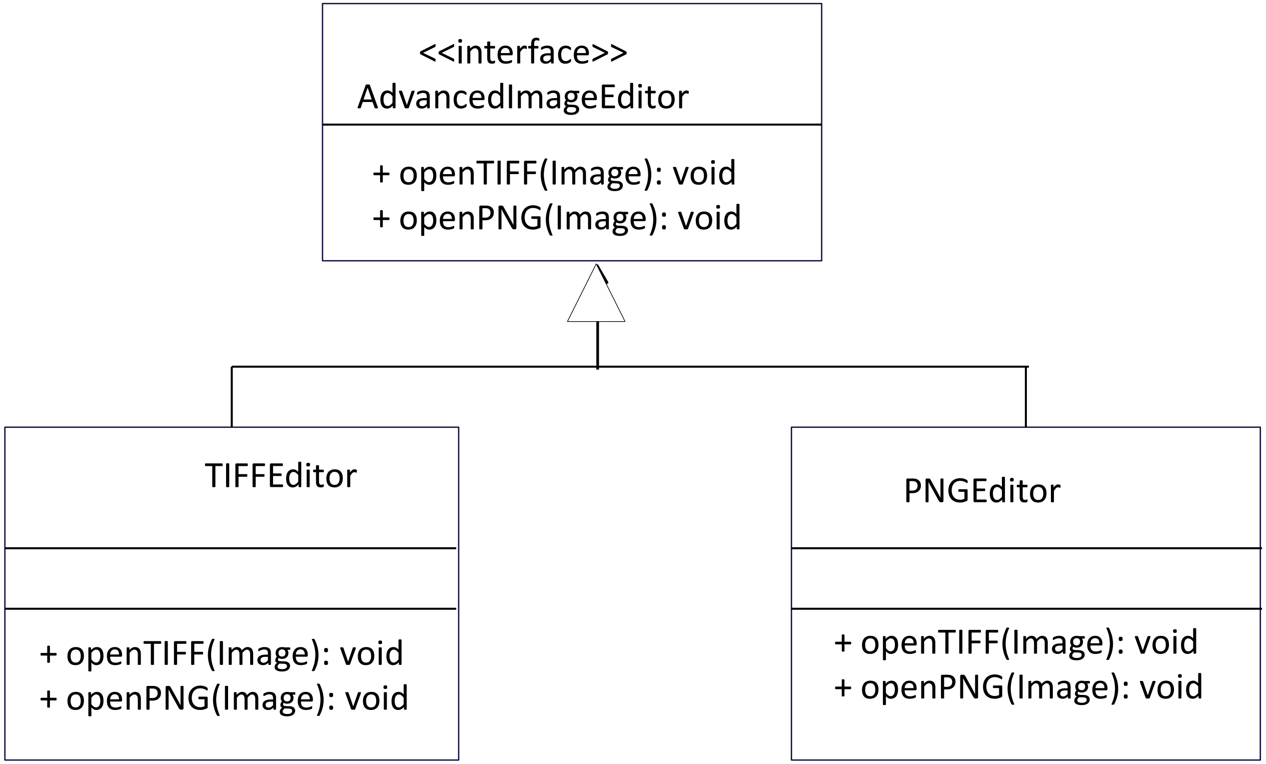


Adapter Pattern

```
public interface ImageEditor{

   public void openImage(Image img);
}

public class JPEGEditor implements ImageEditor{
   public void openImage(Image img)
   {
        System.out.println("Working with JPEG images");
   }

}
```

Adapter Pattern

- There is the third-party library software that can edit and open images of TIFF and PNG format also.



Adapter Pattern

```
public interface AdvancedImageEditor{

   public void openTIFF(Image img);
   public void openPNG(Image img);
}

public class TIFFEditor implements AdvancedImageEditor{
   public void openTIFF(Image img)
   {
         System.out.println("Working with TIFF images");
   }

   public void openPNG(Image img)
   {
         //does nothing
   }

}
```

Adapter Pattern

```
public class PNGEditor implements AdvancedImageEditor{
   public void openPNG(Image img)
   {
         System.out.println("Working with PNGimages");
   }

   public void openTIFF(Image img)
   {
         //does nothing
   }

}
```

Adapter Pattern

- Need to use the TIFFEditor and PNGEditor from within the JPEGEditor.
- Need an Adpater.

- Let’s create ImageAdpater Class, that implements ImageEditor interface.

Adapter Pattern

```
public class ImageAdpater implements ImageEditor{

AdvancedImageEditor imgEditor;

public ImageAdpater(Image img)
{

    if (img.getType().equalsIgnoreCase("TIFF"))
           { imgEditor = new TIFFEditor();}
    else if (img.getType().equalsIgnoreCase("PNG"))
           imgEditor = new PNGEditor();
}

public void openImage(Image img)
{
   if (img.getType().equalsIgnoreCase("TIFF"))  imgEditor.openTIFF(img);
   else if (img.getType(). equalsIgnoreCase("PNG"))  imgEditor.openPNG(img);
}
}
```

Adapter Pattern

```
public class JPEGEditor implements ImageEditor{

ImageAdapter imgAdapt;

   public void openImage(Image img)
   {
         if (img.getType().equalsIgnoreCase(“JPEG”))
           {
            System.out.println("Working with JPEG images");}
             else
                if ((img.getType().equalsIgnoreCase(“TIFF") ||
                 (img.getType().equalsIgnoreCase(“PNG") )))
               {
                imgAdapt = new ImageAdapter(img);
                imgAdapt.openImage(img);
               }

   }
}
```

Adapter Pattern

```
public class AdapterDemo{

public static void main(String[] args)
{

   JPEGEditor imgEditor= new JPEGEditor();

    imgEditor.openImage("img1.jpg");
    imgEditor.openImage("img2.tiff");
    imgEditor.openImage("img3.png”);

}


}
```

Plan ahead…..

Go through Lecture Videos:

- Module 7
- Module 8

Agenda: Lecture 9

- Design Patterns (GOF) (Continued)