Design patterns tutorial point pdf

In software development, the design pattern is a common recurring solution to common software development problems. The design pattern is not a complete design that can be converted directly into code. This is a description or template of how to solve a problem that can be used in different situations. Using design patterns for design patterns can speed up the development process by providing proven, proven development paradigms. Effective software design requires consideration of issues that may become visible only later in implementation. Reusing design patterns helps prevent minor problems that can cause serious problems, and improves code readability for programmers and architects familiar with templates. Often people only understand how to apply certain software design techniques to certain problems. These methods are difficult to apply to a wider range of problems. Design patterns provide common solutions documented in a format that does not require specifics related to a particular problem. In addition, templates allow developers to communicate using well-known, well-understood names for software interactions. Common design patterns can be improved over time, making them more reliable than special projects. Creative Design Patterns These design patterns are all about instant class. This pattern can be further divided into class creation patterns and object creation patterns. While class creation patterns effectively use inheritance in the instant creation process, object creation patterns effectively use delegation to view the work done. Abstract Factory creates a copy of several families of The Builder classes separates the construction object from its presentation Factory method creates a copy of several derivative classes Object Pool Avoid costly acquisition and release of resources by recycling objects that no longer use the prototype fully initiated instance, which will be copied or cloned Singleton class, of which only one instance can exist Structural design patterns These design patterns are all about the class and composition. Structural templates for creating classes use inheritance to compile interfaces. Structural pattern objects determine how to create objects to produce new functionality. Adapter Match interfaces of different bridge classes separates the interface of the object from its implementation Composite structure of the tree of simple and composite objects Decorator Add responsibilities to objects dynamically facade of one class, which is an entire Flyweight Small Grain Copy Used to Effectively Exchange Private Class Data Limits Access to Access to a Proxy That Represents Another Object Behavioral Design Patterns. Behavioral models are the models that are most specifically related to communication Objects. Responsibility Chain The Mode of Request Transfer Between the Command Object Chain Encapsulates The Command Request as an Object Translator The way language elements are included in Iterator Consistently access the elements of the Mediator collection Determines the simplified connection between memento Capture classes and restores the internal state of the Null Object, Intended for action as the default object Observer The way to notify about the change of a number of classes State Alter Object Behavior when its state changes the Strategy Incapulates the algorithm inside the class template method To delay the exact steps of the algorithm in the subclass Visitor identifies a new operation in the class without changing The Concept of Design Patterns has been criticized by some in the field of computer science. The purpose of the wrong problem Need templates is the result of using computer languages or methods with insufficient abstraction ability. With perfect factoring, you should not copy the concept, but simply refer to it. But if something is referenced instead of copying, there is no template for the designation and catalog. Paul Graham writes in the essay Revenge of the Nerds. Peter Norwig makes a similar argument. It demonstrates that 16 of the 23 templates in the design template book (which is primarily focused on THES) are simplified or eliminated (through direct language support) in Lisp or Dylan. The lack of formal grounds for studying design patterns has been overly special, and some argue that the concept is in dire need of a more formal basis. At OOPSLA, 1999, Gang of Four (with full cooperation) was subjected to a show trial, during which they were charged with numerous crimes against computer science. They were convicted by two thirds of the jurors who were present at the trial. Leads to ineffective solutions The idea of a design pattern is an attempt to standardize what has already been adopted by best practices. In principle, this may seem useful, but in practice it often leads to unnecessary duplication of code. It's almost always a more efficient solution for using a well-considered implementation, rather than just barely a good enough design pattern. Not much different from other abstractions Some authors argue that design patterns are not much different from other forms of abstraction, and that using new terminology (borrowed from the architectural community) to describe existing programming phenomena is unnecessary. The Model-View-Controller paradigm is advertised as an example of a pattern that precedes the concept of design patterns for several years. In addition, some argue that the main contribution of the community of design patterns (and the book Gang of Four) was the use of pattern language As a form of documentation; practices that are often ignored are ignored Literature. For more information, diagrams and examples of design patterns, visit our new partner, Refactoring.Guru. 22 design templates and 8 principles explained in depth 406 well-structured, easy-to-use pages without jargon 228 clear and useful illustrations and diagrams Archive with code examples in 4 languages All supported devices: formats EPUB/MOBI/PDF More... Academia.edu no longer supports the Internet Explorer.To browse the Academia.edu and the wider Internet faster and more securely, please take a few seconds to update the browser. Academia.edu uses cookies to personalize content, adapt ads, and improve user experience. Using our website, you agree to our collection of information using cookies. To learn more, check out our Privacy Policy.× Design templates to represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to common problems that software developers face during software development. These solutions have been trial-and-error by numerous software developers over a fairly considerable period of time. What is the Gang of Four (GOF)? In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book called Design Patterns - Elements of Reusable Object-Oriented Software that initiated the concept of pattern design in software development. These authors are known collectively as Gang of Four (GOF). According to these authors, design patterns are primarily based on the following principles of object-oriented design. Program to Interface not implementation Composition Object Favored over Inheritance Using Pattern Design Patterns have two main uses in software development. The common platform for developer design patterns provides standard terminology and is specific to a specific scenario. For example, a monochrome design pattern means using a single object so that all developers familiar with a single design pattern use the same object, and they can tell each other that the program follows monochrome patterns. Best practice design patterns have evolved over a long period of time and provide the best solutions to certain problems that come with software development. Learning these patterns helps inexperienced developers learn software design in a simple and fast way. Types of Design Patterns According to The Design Pattern Design Guide - Elements Object-oriented software, there are 23 design templates that can be classified in three categories: Creative, Structural and Behavioral Models. We'll also discuss another design template category: J2EE design patterns. S.N.Pattern - Description 1Creational PatternsThese design patterns provide a way to create objects while hiding the logic of creating rather than instantaneous use of objects directly with the help of a new operator. Operator. gives the program more flexibility in deciding which objects should be created for the case. 2Structural patternsTech design patterns relate to the class and composition of the object. The concept of inheritance is used to create interfaces and identify ways to create objects to gain new functionality. 3Behavioral Patterns These design patterns are specifically related to communication between objects. 4J2EEThese design patterns are specifically related to the level of presentation. These patterns are identified by the Sun Java Center. Design patterns are best practices used by experienced object-oriented software developers. Design patterns are solutions to common problems that software developers face during software development. These solutions have been trial-and-error by numerous software developers over a fairly considerable period of time. What is the Gang of Four (GOF)? In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book called Design Patterns - Elements of Reusable Object-Oriented Software that initiated the concept of pattern design in software development. These authors are known collectively as Gang of Four (GOF). According to these authors, design patterns are primarily based on the following principles of object-oriented design. Program to Interface not implementation Composition Object Favored over Inheritance Using Pattern Design Patterns have two main uses in software development. The common platform for developer design patterns provides standard terminology and is specific to a specific scenario. For example, a monochrome design pattern means using a single object so that all developers familiar with a single design pattern use the same object, and they can tell each other that the program follows monochrome patterns. Best practice design patterns have evolved over a long period of time and provide the best solutions to certain problems that come with software development. Learning these patterns helps non-experienced developers learn software design in a simple and quick way. Design Pattern Types Consistent with Pattern Design Design Patterns - Elements of reusable object-oriented software, there are 23 design patterns. These models can be categorized into three categories: baptism, structural and behavioral models. We'll also discuss another category of design templates: J2EE design patterns. S.N.Pattern - Description These design patterns provide a way to create objects by hiding the logic of creation rather than instantaneous use of objects directly with the help of a new operator. This gives the program more flexibility in deciding which objects should be created for the case. 2Structural patternsTech design patterns relate to the class and composition of the object. The concept of inheritance is used to compile interfaces and ways to create objects to get new functionality. 3Behavioral Patterns These design patterns are specifically related to communication between objects. 4J2EEThese design patterns are specifically related to the level of presentation. These patterns are identified by the Sun Java Center. The Factory Pattern Factory template is one of the most commonly used design patterns in Java. This type of design pattern is under the creation pattern because this pattern provides one of the best ways to create an object. In the Factory template, we create an object without exposing the customer to the creation logic, and refer to a newly created object using a common interface. Implementation We are going to create a form interface and specific classes of implementation of the Shape interface. The next step is the ShapeFactory factory class. FactoryPatternDemo, our demo class will use ShapeFactory to get the object shape. It will transmit information (CIRCLE / RECTANGLE / SQUARE) to ShapeFactory to get the type of object that it needs. Step 1 Create an interface. Form. Java Public Form interface and void draw (); Step 2 Create specific classes by implementing the same interface. Rectangle.java public-class Rectangle implements a form - @Override public void draw () - System.out.println (Inside the rectangle::d raw() method.); Square.java Public Class Square implements a form - @Override public draw of emptiness () - System.out.println (Inside the square::d raw()method.); Circle.java Public Class Circle implements a form @Override public void draw () - System.out.println (Inside the Circle::d raw() method.); Step 3 Create a factory to create an object of a particular class based on this information. ShapeFactory.java public class ShapeFactory /use getShape method to get an object type form form getShape (String shapeType) if (shapeType - null) return is zero; - if (shapeType.equalsIgnoreCase (CIRCLE) - the return of the new Circle (); - even if (formType.equalsIgnoreCase (RECTANGLE) ; Step 4 Use the factory to get an object of a particular class by passing information such as type. FactoryPatternDemo.java public class FactoryPatternDemo - public static emptiness of the core (String) - ShapeFactory shapeFactory - new ShapeFactory (); get the Circle object and call it the toss method. Form1 - shapeFactory.getShape (CIRCLE); Circle shape1.draw call draw method get the Rectangle object and call it the toss method. Form2 - shapeFactory.getShape (RECTANGLE); Rectangle shape2.draw call draw method Get the object of the square and call it the method of tossing. Form Form3 - shapeFactory.getShape Square-form tossing method3.draw(); Step 5 Check the output. Inside the circle::d Method. Inside the rectangle::d method. Inside the square: :d method. Design pattern - - Factory Patterns Abstract Factories work around a super-factory that creates other plants. This plant is also called the factory factory. This type of design pattern is under the creation pattern because this pattern provides one of the best ways to create an object. In the Factory Abstract template, the interface is responsible for creating a factory of related objects without directly specifying their classes. Each factory generated can produce objects according to the Factory template. Implementation We are going to create a shape interface and a specific class by implementing it. The next step is to create an abstract factory class AbstractFactory. The ShapeFactory factory class, which expands AbstractFactory, is defined. FactoryProducer is the creator/generator of the factory. AbstractFactoryPatternDemo, our demo class uses FactoryProducer to get AbstractFactory. It will transmit information (CIRCLE/RECTANGLE/SQUARE for Form) AbstractFactory to get the type of object it needs. Step 1 Create an interface for forms. Form. Java Public Form interface and void draw (); Step 2 Create specific classes by implementing the same interface. RoundedRectangle.java public-class RoundedRectangle implements the form - @Override public draw of emptiness () - System.out.println (Inside the roundedRectangle::d raw() method.); RoundedSquare.java public class RoundedSquare implements form - @Override public draw of emptiness () - System.out.println (Inside RoundedSquare::d raw() method.); Rectangle.java public-class Rectangle implements a form - @Override public void draw () - System.out.println (Inside the rectangle::d raw() method.); Step 3 Create an abstract class to get factories for objects of normal and rounded shape. AbstractFactory.java public abstract class AbstractFactory - abstract form getShape (String shapeType) ; Step 4 Create Factory classes that expand AbstractFactory to create a class-specific object based on this information. ShapeFactory.java public class ShapeFactory expands AbstractFactory - @Override public form getShape (String shapeType) if (shapeTypeType.equalsIgnoreCase) still, If (formType.equalsIgnoreCase) - RoundedShapeFactory.java Public Class RoundedShapeFactory Expands AbstractFactory - @Override public form getShape (String shapeType) if (shapeTypeType.equalsIgnoreCase)) still if (shapeType.equalsIgnoreCase) still if (shapeType.equalsIgnore) (shapeType) return a new Rounded Step 5 Create a generator factory/producer class to get the plants by passing information such as the Form FactoryProducer.java public-class FactoryProducer - public static AbstractFactory getFactory (boolean rounded) if (rounded) bring back the new RoundedShapeFactory (); yet to bring back the new ShapeFactory Step 6 FactoryProducer to get AbstractFactory's to get plants of specific classes by passing information such as type. AbstractFactoryPatternDemo.java public class AbstractFactoryPatternDemo - public static emptiness core (String) - /get the form of factory AbstractFactory shapeFactory - FactoryProducer.getFactory (false); Get a rectangular shape shape entity1 - shapeFactory.getShape (RECTANGLE); Rectangle shape1.draw Get an object shaped like a square shape2 - shapeFactory.getShape call draw method of Shape Square shape2.draw/get a rectangular shape shape-shaped object3 - shapeFactory1.getShape (RECTANGLE); Shape Rectangle shape3.draw Get the square Shape4 shape object - shapeFactory1.getShape Shape Square shape4.draw.) Step 7 Check the exit. Inside the rectangle::d method. Inside the square: :d method. Inside RoundedRectangle: :d raw() method. Inside RoundedSquare::d raw() method. The Singleton Pattern Singleton template is one of the simplest design patterns in Java. This type of design pattern is under the creation pattern because this pattern is one of the best ways to create an object. This pattern includes one class that is responsible for creating your own object, making sure that only one object is created. This class provides access to its only object that can be accessed directly without the need for instantaneous introduction of the class object. The SingleObject class has its designer as a private and has a static copy. SingleObject class provides a static method to get its static instance to the outside world. SingletonPatternDemo, our demo class will use the SingleObject class to get a SingleObject object. Step 1 Create a singleton class. SingleObject.java public class SingleObject /create a singleObject private static one-object copy - the new SingleObject (); make the designer closed, so that this class can not be //instantiated private SingleObject ()/ / Get the only object available to the public static SingleObject getInstance ()) reverse instance; Step 2 Get the only object from the singleton class. SingletonPatternDemo.java public class SingletonPatternDemo - public static emptiness core (String) / //Illegal design // Compilation time error: SingleObject designer () is not visible //SingleObject object - new SingleObject (); Get the only SingleObject Step 3 available to check the exit. Hello world! The Builder Pattern Builder creates a complex object using simple objects and using a step-by-step approach. This type The pattern fits the creation pattern because this pattern provides one of the best ways to create an object. The Builder class builds the last object step by step. This builder is not dependent on other facilities. Implementation We reviewed the business case of a fast food restaurant, where a typical meal can be a hamburger and a cold drink. The burger can be either a Veg Burger or a chicken burger and will be packed with wrappers. A cold drink can be either coke or pepsi and will be packed into a bottle. We're going to create an interface item representing food items such as burgers and cold drinks and specific classes implementing the item interface and packaging interface, representing food packaging and specific classes implementing the packaging interface, as the burger will be packaged in a wrapper and a cold drink will be packaged in a bottle. We then create a Nutrition Class with ArrayList item and MealBuilder to create different types of power

facility by combining the item. BuilderPatternDemo, our demo class will use MealBuilder to create food. Step 1 Create an interface item representing food and packaging. Item's public interface item is the public name for the line. Public packaging packaging Public float price Packing.java public interface packaging is a public line package Step 2 Create concreate classes by implementing the Packaging interface. Wrapper.java public class Wrapper implements packaging @Override public line package () - Return Wrapper; The Bottle.java public-class bottle sells @Override a public line package - Bottle return; Step 3 Create abstract classes by implementing an element interface that provides the default functionality. Burger.java public abstract class Burger implements item no @Override public packaging packaging () - return of the new wrapper (); - @Override public abstract price of the float (); ColdDrink.java public abstract class ColdDrink implements the item @Override public packaging packaging () - the return of a new bottle (); - @Override public abstract price of the float (); Step 4 Create specific classes of Burger expansion and ColdDrink classes VegBurger.java public class VegBurger expands Burger - @Override public float price () - return 25.0f @Override; ChickenBurger.java public class ChickenBurger expands Burger - @Override price of public swimming () - a return of 50.5f @Override; Coke.java public-class Coke is distributed by ColdDrink @Override the public float price () - a return of 30.0f @Override; Pepsi.java public class Pepsi expands ColdDrink @Override price of public swimming () - return 35.0f @Override; Step 5 Create a Nutrition class with the items of the item above. Meal.java import java.util.ArrayList; Import питание государственного класса -<Item> частные предметы перечня - новый ArrayList<Item>();</Item> </Item> </Item> inespidacies addItem (item) ' items.add (detail); - getCost public float ()) float costing 0.0f; For (item item: items) - item.price()item.name; System.out.print (, Packaging : zuit.packing ().package ()); System.out.println (price: zyamak.price); Step 6 Create a MealBuilder class, the actual class of builders responsible for creating Meal objects. MealBuilder.java-public-class MealBuilder meal.addItem meal.addItem (new Coca-Cola);; Step 7 BuiderPatternDemo uses MealBuider to demonstrate the builder's template. BuilderPatternDemo.java public class BuilderPatternDemo - public static void core (String) - MealBuilder mealBuilder - new MealBuilder (); VegMeal Nutrition - mealBuilder.prepareVegMeal System.out.println vegMeal.showItems(); System.out.println (Total cost: vegMeal.getCost); nonVegMeal Nutrition - mealBuilder.prepareNonVegMeal (); System.out.println (Не-Вег Питание); nonVegMeal.showItems(); System.out.println Step 8 Check the exit. Veg Nutrition Item : Veg Burger, Packaging : Wrapping, Price : 25.0 Item : Coca-Cola, Packaging : Bottle, Price: 30.0 Total Cost: 55.0 Non-Veg Meal Item : Chicken Burger, Packaging: Wrap, Price : 50.5 Item : Pepsi, Packaging : Bottle, Price: 35.0 Total Cost: 85.5 Prototype Pattern Pattern This type of design pattern is under the creation pattern because this pattern is one of the best ways to create an object. This pattern includes the implementation of a prototype interface that says to create a clone of the current object. This pattern is used when creating an object directly is expensive. For example, an object must be created after an expensive database operation. We can cache an object, return its clone at the next request, and update the database as needed, thereby reducing database calls. Implementation We are going to create an abstract form class and specific form expansion classes. The ShapeCache class is defined as the next step that stores the form objects in Hashtable and returns their clone on request. PrototypPatternDemo, our demo class will use the ShapeCache class to get the Shape object. Step 1 Create an abstract class by implementing a clonable interface. Shape.java Public Abstract Class Form implements Cloned - private string ID; Protected line type abstract void to draw this.id catch (CloneNotSupportedException e) - e.printStackTrace Step 2 Create specific classes that expand the above class. Rectangle.java public class Rectangle extends the shape - public rectangle () - rectangle type; - @Override public draw of emptiness () - System.out.println (Inside the rectangle::d raw() method.); Square Square Square's public class square expands the shape - public square - type square; - @Override public draw of emptiness () - System.out.println (Inside the square::d raw() method.); Circle Circle Circle Circle Expands Form - Community Circle - Circle Type; - @Override public draw of emptiness () - System.out.println (Inside the circle::d raw() method.); Step 3 Create a class to get the classes to concreate from the database and store them in hashstable. ShapeCache.java import java.util.Hashtable; ShapeCache Public Class - Private Static Form, HashtableMap - the new Hashtable Public Static Form, Shape-and-GetShape -- ShapeMap.get (shapeId); Return (form) cachedShape.clone Each form is requested by a database and a form is created // shapeMap.put (shapeKey, form); For example, we add three figures of the public static void loadCache () - Circle - a new circle (); circle.setId(1); shapeMap.put(circle.getId); Square square - new square (); square.setId (2); shapeMap.put The rectangle is a new rectangle. rectangle.setId (3); shapeMap.put (rectangle.getId); Step 4 PrototypePatternDemo uses the ShapeCache class to get clones of shapes stored in hashstable. PrototypePatternDemo. Java public class PrototypePatternDemo - public static emptiness core (String) - ShapeCache.loadCache (); Form of a cloned step (form) ShapeCache.getShape (1); System.out.println (Form - clonedShape.getType(); Form of Cloned Shop2 (form) ShapeCache.getShape (2); System.out.println (Form - clonedShape2.getType;); Form clonedShape3 (form) ShapeCache.getShape (3); System.out.println Step 5 Check the exit. Form : Circle Shape : Square form : The rectangle adapter adapter template works as a bridge between two incompatible interfaces. This type of design pattern fits the structural pattern because it combines the capabilities of two independent interfaces. This template includes a single class that is responsible for attaching functions to independent or incompatible interfaces. An example of real life would be the case of a card reader that acts as an adapter between a memory card and a laptop. You connect the memory card to the card reader and card reader to your laptop so that the memory card can read with a laptop. We demonstrate the use of the Adapter template through the following example, in which the audio player device can play mp3 files only and wants to use an advanced audio player capable of playing vlc and mp4 files. </String,></String,> </String,></String,> We have the MediaPlayer Interface interface and a specific class of AudioPlayer implementation interface MediaPlayer. AudioPlayer can play mp3 format audio files by default. We have a different AdvancedMediaPlayer interface and specific AdvancedMediaPlayer implementation classes. These classes can play vlc and mp4 file format. We want AudioPlayer to play in other formats as well. To achieve this goal, we have created a MediaAdapter-class adapter that implements the MediaPlayer interface and uses AdvancedMediaPlayer objects to play the format required. AudioPlayer uses a MediaAdapter-class adapter, giving it the desired type of sound without knowing the actual class that can be played in the right format. AdapterPatternDemo, our demo class will use the AudioPlayer class to play different formats. Step 1 Create interfaces for media player and advanced media player. MediaPlayer.java public MediaPlayer interface - public void of playback (String audioType, String fileName); AdvancedMediaPlayer.java public interface AdvancedMediaPlayer - public void playVlc (String fileName); public emptiness playMp4 (String fileName); Step 2 Create specific advancedMediaPlayer implementation classes. VlcPlayer.java public-class VlcPlayer implements AdvancedMediaPlayer @Override public void playVlc (String fileName) - System.out.println (Game vlc file. Name: File name); @Override public void playMp4 (String fileName) @Override/do nothing - Mp4Player.java public class Mp4Player implements AdvancedMediaPlayer @Override public void playVlc (String fileName) - /do nothing @Override public void playMp4 (StringName) - System.out.println Step 3 Create a MediaPlayer implementation class. MediaAdapter.java public class MediaAdapter implements MediaPlayer - AdvancedMediaPlayer advancedMusicPlayer; Public MediaAdapter (String audioType) if (audioType.equalsIgnoreCase (vlc)) - still if (audioType.equalsIgnoreCase (mp4)) - advancedMusicPlayer - the new Mp4Player (); - @Override public invalid playback (String audioType, String fileName) - if (audioType.equalsIgnoreCase)) still if (audioType.equalsIgnoreCase (mp4) advancedMusicPlayer.playMp4 (fileName); Step 4 Create a specific media interface implementation class. AudioPlayer.java public-class AudioPlayer implements MediaPlayer - MediaAdapter mediaAdapter; @Override public invalid game (String audioType, String fileName) - /embedded support for music playback mp3 if (audioType.equalsIgnoreCase (mp3) . Name: File name); mediaAdapter supports the playback of other file formats if (audioType.equalsIgnoreCase (vlc) (mp4)) , mediaAdapter - новый MediaAdapter (audioType); mediaAdapter.play (audioType, fileName); fileName); else' System.out.println (Invalid Media. AudioType and format is not supported); Step 5 Use AudioPlayer to play different types of audio formats. AdapterPatternDemo.java public class AdapterPatternDemo - public static emptiness core (String) - AudioPlayer audioPlayer - new AudioPlayer (); audioPlayer.play (mp3, beyond the horizon.mp3); audioPlayer.play (mp4, alone.mp4); audioPlayer.play (vlc). Step 6 Check the exit. Play the mp3 file. Title: Beyond horizon.mp3 Play mp4 file. Title: alone.mp4 Play vlc file. Title: Far far.vlc Disabled Media. The avi format, not supported by Bridge Pattern Bridge, is used where we need to separate abstraction from implementation so that they can change independently from each other. This type of design pattern falls under a structural pattern because this pattern separates the implementation class and the abstract class, ensuring the structure of the bridge between them. This template includes an interface that acts as a bridge that makes the functionality of specific classes independent of the interface implementation classes. Both types of classes can be changed structurally without affecting each other. We demonstrate the use of the Bridge pattern through the following example, in which a circle can be drawn in different colors using the same abstract class method, but different classes of bridge performers. Implementation We have a DrawAPI interface that acts as a bridge of implementation and specific classes Of RedCircle, GreenCircle implementation interface DrawAPI. The form is an abstract class and will use the DrawAPI object. BridgePatternDemo, our demo class will use the Shape class to draw different color circles. Step 1 Create a bridge implementation interface. DrawAPI.java public interface DrawAPI - public void drawCircle (radius int, int x, int y); Step 2 Create concrete bridge implementation classes by implementing the DrawAPI interface. RedCircle.java public class RedCircle implements DrawAPI - @Override public void drawCircle (radius int, int x, int y) - System.out.println (Drawing circle color: red, radius: radius, x: X, I y) . GreenCircle.java public class GreenCircle implements DrawAPI - @Override public void drawCircle (radius int, int x, int y) - System.out.println (Drawing a circle color: green, radius: radius x X , I Step 3 Create an abstract class shape using the DrawPIA interface. Shape.java Public Abstract Form Class - Protected DrawAPI drawAPI; Protected Form (DrawAPI drawAPI) - this.drawAPI - drawAPI; Public abstract void draw Step 4 Create a specific class by implementing the Shape interface. Circle Circle Circle Circle expands the shape - private int x, y, radius; public circle (int x, int y, int radius, drawAPI) - супер (drawAPI); this.x x; this.y и y; this.radius - радиус; радиус; Step 5 Use Form and DrawAPI classes to draw different color circles. BridgePatternDemo.java public class BridgePatternDemo - public static void core (String) - Form redCircle - new circle (100 100, 10, new RedCircle()); Форма greenCircle - новый круг (100,100, 10, новый GreenCircle(); redCircle.draw(); greenCircle.draw(); Шаг 6 Проверка вывода. Рисунок Круг цвета: красный, радиус: 10, x: 100, 100 Рисование Круг цвет: зеленый, радиус: 10, x: 100, 100 Фильтр / Критерии шаблон фильтр шаблон шаблон шаблон шаблона шаблон шаблона является шаблоном дизайна, который позволяет разработчикам фильтровать набор объектов, используя различные критерии, приковывания их в разъединение образом через логические операции. Этот тип шаблона проектирования под подходит под структурный шаблон, поскольку эта модель сочетает в себе несколько критериев для получения единых критериев. Реализация Мы собираемся создать объект person, интерфейс Критерии и конкретные классы, реализуя этот интерфейс для фильтрации списка объектов Person. CriteriaPatternDemo, our demo class uses Criteria objects to filter a list of Person objects based on different criteria and their combinations. Step 1 Create a class where criteria should apply. Person.java public-class Face - private name of the string; Private string floor; private line maritalStatus; public person (String name,String gender, String maritalStatus) - this.name - name; this.gender - gender; this.maritalStatus - maritalStatus; Step 2 Create an interface for criteria. Criteria.java import java.util.List; The public list is in line with the list. Step 3 Create specific classes by implementing the Criterion interface. CriteriaMale.java import java.util.ArrayList; import java.util.List; The CriteriaMale Public Class implements the Criteria - @Override public list corresponds toCriteria (List of Persons) - List of Men Personna - New ArrayList (); for (Face Face: Face) - if the person is a java.util.ArrayList. import java.util.List; The CriteriaFemale Public Class implements the Criteria - @Override public list meetsCriteria (List of Persons) - List of Women Personas - New ArrayList (); for (Face Face: Face) - if the person is a person, java.util.ArrayList; import java.util.List; The CriteriaSingle Public Class implements the Criteria: @Override public list of persons is compliant with the List of Persons - List of Persons for (Face Face: Face) - if (person.getMaritalStatus (.equalsIgnoreCase </Person> </Person> </Person> </Person> </Person> </Person> </Person> </Person> </Person> </Person> </Person> </Person> </Person> - return of the singlePersona; - AndCriteria.java import java.util.List; AndCriteria's public class implements the Criteria - private criteria; Private Criteria otherCriteria; Public AndCriteria (Criteria criteria, criteria of otherCriteria) - this.criteria - criteria; this.otherCriteria - otherCriteria; - @Override public List'lt;meetCriteria (List of persons) - List of firstCriteriaPersons OrCriteria.java java import Java.util.List; AndCriteria's public class implements the Criteria - private criteria; Private Criteria otherCriteria; Public AndCriteria (Criteria criteria, criteria of otherCriteria) - this.criteria - criteria; this.otherCriteria - otherCriteria; - @Override public list of persons (persons) List otherCriteriaItems - otherCriteria.meetCriteria (faces); For (Face Face : otherCriteriaItems) - if (!firstCriteriaItems.contains (person)) Step4 Use different criteria and their combination to filter people out. CriteriaPatternDemo.java import java.util.ArrayList; import java.util.List; CriteriaPatternDemo Public Class - Public Static Void Core (String) - persons.add (new face (Robert, Man, Lonely); (John, Man, Married); women.add (new face (Laura, woman, married)) women(add); persons.add (Mike, Male) Male Criteria and New CriteriaMeil (); Criteria of Women and New CriteriaFemale (); Criteria Uniform - New CriteriaSingle (); SingleMale Criteria - New AndCriteria (single, male); Criteria singleOrFemale System.out.println (Men:); printPersons (men.meetCriteria (faces); System.out.println (Women:); printPersons (woman.meetCriteria (faces)); System.out.println (Single Men:); printPersons (singleMale.meetCriteria); System.out.println (Singles or Women:); printPersons (singleOrFemale.meetCriteria (persons))); - public static void of printPersons (Face List) for (face: faces: persons) - System.out.println (Face : - person.getName () , Gender: Step 5 Check Out. Men: Face: Name: Robert, Gender: Men, Marital Status : Lonely : : Name: John, Paul: Men, Gender: Men, Gender: Men, Gender: Men' Status: Men, Family Status: Men, Marital Status: Married: Married: : Name : Name: Men, Family Position:: Name : Diana, Paul : Female, Marital Status : Single Men: Face : : Name : Robert, Paul : Men, Marital Status : Single : Name : Name : Name : Name: Men, Marital status : Single : Name : Name : Bobby, Paul : Male, Marital Status : Single Paul : Men, Marital Status : Lonely : Lonely : Name : Name : Diana, Gender : Woman, Marital Status : Lonely : Lonely : Name : Name : Name : Men, Marital Status : Lonely : Composite This type of design pattern is under a structural pattern because this pattern creates the structure of the tree of a group of objects. This template creates a class that contains a group of objects of its own. This class provides ways to change a group of the same objects. We demonstrate the use of the Composite template in the following example, which shows the hierarchy of your organization's employees. Implementation We have a Class Employee that acts as a composite type of actor template. CompositePatternDemo, our demo class will use the staff class to add department level hierarchy and print to all employees. Step 1 Create an employee class with a list of employee objects. Employee.java import java.util.ArrayList; import java.util.List; State Class Employee - private line name; Private line department Private int salary; Private subordinates constructor of public servant (String name, String dept, int sal) - this.name - name; this.dept - dept; this.salary - sal; subordinates - the new ArrayList - Public void to add (Employee e) - subordinates.add (e); - Public void to remove employees (employee e) - subordinates. Step 2 Use an employee class to create and print a hierarchy of employees. CompositePatternDemo.java public class CompositePatternDemo - Public Static Void Core (String) - Employee CEO - New Employee (John, CEO, 30,000); Head of Employee Operations and a New Employee (Robert, Head Sales, 20,000); Head of Employee Marketing and New Employee (Michelle, Chief Marketing Officer, 20,000); Clerk 1 - new employee (Laura, Marketing, 10,000); Clerk2 employee - new employee (Bob, Marketing, 10,000); Sales of employeesExecutive1 - new employee (Richard, Sales, 10,000); Sales of employeesExecutive2 - a new employee (Rob,Sales, 10,000); CEO.add CEO.add headSales.add headSales.add headMarketing.add (clerk1); </Employee></Employee></Employee>Print all System.out.println (CEO) employees for (Employee headEmployee : CEO.getSubordinates () - System.out.println (headEmployee); For (Employee: headEmployee.getSubordinates() - System.out.println (employee); Step 3 Check the output. Employee : Name : John, Department : CEO, salary : 30,000 : Name : Robert, Department : Chief of Sales, Salary : 20,000 : Name: Richard, Department : Sales, salary : 10,000 department : Sales, salary : 10,000 : Employee : Name : Michelle, Department : Chief Marketing, Salary : 20,000 : Name : Name : Laura, Department : Marketing, Salary : 10,000 : Employee : Name: Bob, Department : Marketing, Salary:10,000 - Decorator Pattern Decorator Pattern Pattern Allows you to add new functionality to the existing facility without changing the structure of its structure. This type of design pattern fits the structural pattern because this pattern acts as a wrapper to an existing class. This template creates a decorator class that wraps the original class and provides additional functionality while keeping the signature class methods intact. We demonstrate the use of the Decorator pattern through the following example, in which we decorate the shape with color without changing the shape of the class. We then create an abstract class of ShapeDecorator decorators, implementing the Shape interface and having the Shape object as a variable instance. RedShapeDecorator is a concrete class sold by ShapeDecorator. DecoratorPatternDemo, our demo class will use RedShapeDecorator to decorate the objects of the form. Step 1 Create an interface. Form. Java Public Form interface and void draw (); Step 2 Create specific classes by implementing the same interface. Rectangle.java public-class Rectangle implements a form - @Override public void draw () - System.out.println (Form: Rectang); Circle Circle Community Class class implements a form @Override public draw of emptiness () - System.out.println (Form: Circle); Step 3 Create an abstract decorator class by implementing the Shape interface. ShapeDecorator.java public abstract class ShapeDecorator implements a form - a protected form decorated withShape; Public ShapeDecorator (Shape decoratedShape); Step 4 Create a class of concrete decorators that expands the ShapeDecorator class. RedShapeDecorator.java public-class RedShapeDecorator expands ShapeDecorator - public RedShapeDecorator (Shape decoratedShape) - super (decoratedShape); - @Override public draw of emptiness () - decoratedShape.draw (); setRedBorder (decorated shape); Step 5 Use RedShapeDecorator to decorate the shapes. DecoratorPatternDemo.java public class DecoratorPatternDemo - Public static void core (String) Circle - a new circle Form redCircle - new RedShapeDecorator (new circle)); The form of redRectangle is the new RedShapeDecorator (new rectangle); System.out.println circle.draw(); System.out.println redCircle.draw(); System.out.println redRectangle.draw(); Step 6 Check output. Круг с нормальной пограничной формой: Круг круг красной пограничной формы: Круг пограничный цвет: Красный прямоугольник красной пограничной формы: Прямоугольный пограничный цвет: Красный фасад шаблон Фасад шаблон шаблона шаблона скрывает сложности системы и обеспечивает интерфейс для клиента, с помощью которого клиент может получить доступ к системе. This type of design pattern is under a structural pattern because this pattern adds an interface to the system's output to hide its complexities. This model includes a single class that provides simplified methods that are required by the client and delegates to call to existing methods of system classes. Implementation We are going to create a form interface and specific classes of implementation of the Shape interface. The next step is to determine the ShapeMaker facade class. The ShapeMaker class uses specific classes to delegate users to these classes. FacadePatternDemo, our demo class will use the ShapeMaker class to show results. Step 1 Create an interface. Form. Java Public Form interface and void draw (); Step 2 Create specific classes by implementing the same interface. Rectangle.java public-class Rectangle implements a form - @Override public draw of emptiness () - System.out.println (Rectangle::d raw()); Square.java Public Class Square implements the form - @Override public draw of the void () - System.out.println (Square::d raw)); Circle Circle Community Circle's community class implements a form @Override a public void draw - System.out.println (Circle::d raw()); Step 3 Create a facade class. ShapeMaker.java public class ShapeMaker - private range of forms; A private rectangle in shape The private shape of the square Public ShapeMaker - Circle - New Circle rectangle - new rectangle Square - new square Step 4 Use the facade to draw different types of shapes. FacadePatternDemo.java public class FacadePatternDemo - public static void core (String) - ShapeMaker shapeMaker - new ShapeMaker (); shapeMaker.drawCircle (); shapeMaker.drawRectangle(); shapeMaker.drawSquare (); Step 5 Check the exit. Circle::d raw() Rectangle::d raw() Square::d raw () Flyweight Pattern Flyweight pattern is primarily used to reduce the number of objects created to reduce memory footprint and improve performance. This type of design pattern is under a structural pattern because this template provides ways number of objects, thereby improving the structure of the required application objects. Flyweight template trying to reuse already similar objects, saving them and creating a new object when the appropriate object is not found. We'll demonstrate this pattern by drawing 20 laps of different locations, but we only create 5 objects. Only 5 colors are available, so the color property is used to test existing Circle objects. Implementation We are going to create a form interface and a specific class circle implementation form interface. The next step is the ShapeFactory factory class. ShapeFactory have a HashMap circle having a key as the color of the Circle object. Whenever a request comes to create a circle of a certain color ShapeFactory. ShapeFactory checks the circle object in its HashMap, if the Circle object is found, that object is returned otherwise a new object is created, stored in a hash card for further use and returned to the customer. FlyWeightPatternDemo, our demo class will use ShapeFactory to get the object shape. It will transmit information (red/green/blue/black/white/white) to ShapeFactory to get the circle of desired color it needs. Step 1 Create an interface. Form. Java Public Form interface and void draw (); Step 2 Create a specific class by implementing the same interface. Circle.java public-class Circle implements a form - a private string color; Private int x; Private int y; Private radius int; (String color) - this.color - color; - public invalid setX (int x) - this.x x; - public emptiness setY (int y) - this.y - y; - public emptiness setRadius (radius int) - this.radius - radius; - @Override public void is a draw () - System.out.println (Circle: Draw() Step 3 Create a factory to create an object of a particular class based on this information. ShapeFactory.java import java.util.HashMap; ShapeFactory public class / Uncomment compiler directive line and / Javac s.java will compile correctly. @SuppressWarnings (unchecked) private static final Circle HashMap - the new HashMap Public static form getCircle (String Color) - Circle Circle (Circle)circleMap.get (color); System.out.println (Creating a circle of color: reverse circle; Step 4 Use the factory to get an object of a particular class by passing information such as color. FlyweightPatternDemo.java public class FlyweightPatternDemo - private static final colors of strings . Public Static Void Basic (String) - for (int i'0; i zlt; 20; qy) - Circle Circle (Circle)ShapeFactory.getCircle (getRandomColor()); circle.setX (getRandomX()); circle.setY (getRandomY()); circle.setRadius (100); circle.draw (); - private static line getRandomColor (int) (Math.random.) Step 5 Check the output. Creating a circle of color : Black Circle: Draw () Color : Black, x : 36, y :71, Radius :100 Creating a circle of color : Green circle: Draw () Color : Green, x : 27, y :27, radius :100 Creation circle of color : White circle: Draw () Color: White, x: 64, y :10, radius :100 Creation of a circle of color : Red circle: Draw () Color: Red, x : 15, y:44, radius:100 Circle: Draw () Color: Green, x: 19, Radius :100 Circle: Draw () Color: Green, x : 94, y:32, Radius :100 Circle: Draw: White, x : 69, y :98, Radius :100 Creating a circle of color : Blue Circle Draw () Color: Blue, Draw x : 13, y:4, radius :100 Circle: Draw : Color: Green, x: 21, y :21, radius :100 Circle: Draw:100 Color: Blue, x: 55, y :86, radius :100 Circle : Draw () Color: White, x : 90, y :70, Radius :100 Circle: Draw () Color: Green, x : 78, y :3, radius :100 Circle: Draw () Color: Green, x: 64, y :89, radius :100 Circle: Draw () Color: Blue, x: 3, y :91, radius :100 Circle: Draw:90: Colour: Blue, x: 62, y :82, Radius :100 Circle: (97) y :61, radius :100 Circle: Draw () Color: Green, x : 86, y:12, radius :100 Circle: Draw () Color: Green, x : 38, y :93, radius :1 Circle: Draw () Color : Red, x : 76, y:82, radius:100 Circle: Draw () Color: Blue, x: 95, y :82, radius :100 proxy pattern in the proxy pattern, class represents the functionality of another class. This type of pattern design approached the structural scheme. In the Proxy template, we create an object that has the original object to interface its functionality with the outside world. Implementation We are going to create an image interface and specific classes to implement the image interface. ProxyImage is a proxy class for memory loss when loading RealImage objects. ProxyPatternDemo, our demo class will use ProxyImage to get the image object to download and display as it needs. Step 1 Create an interface. Image.java Public Interface Image - invalid display (); Step 2 Create specific classes by implementing the same interface. RealImage.java public class RealImage implements the image - private line fileName; Public RealImage (String fileName) - this.fileName - fileName; loadFromDisk (fileName); - @Override public invalid display () - System.out.println (Display - FileName); Private void loadFromDisk (String fileName) System.out.println (Download - FileName); ProxyImage.java public class ProxyImage sells the image of private RealImage realImage; Private string fileName; Public ProxyImage (String fileName) - this.fileName - fileName; - @Override public display of emptiness () - if (realImage) - null) - realImage - the new RealImage (fileName); Step 3 Use ProxyImage to get a RealImage class object when needed. ProxyPatternDemo.java ProxyPatternDemo - Public Static Void Core (String) - Image - new ProxyImage (test_10mb.jpg); (test_10mb.jpg); Will be downloaded from image.display System.out.println (); The image will not be downloaded from image.display Step 4 Check the output. Download test_10mb.jpg Display test_10mb.jpg Display test_10mb.jpg Chain responsibility Pattern As the title suggests, the chain of responsibility pattern creates a chain of receiver objects for query. This pattern separates the sender and the recipient of the request, depending on the type of query. This model is under threat of behavior. In this template, each receiver usually contains a link to a different receiver. If one object can't handle the request, it passes the same to the next receiver and so on. Implementation We created an abstract class AbstractLogger with a registration level. We then created three types of abstractLogger extension loggers. Each registrar checks the level of the message to its level and prints accordingly, otherwise does not print or transmit the message to their next registrar. Step 1 Create an abstract logger class. AbstractLogger.java public abstract class AbstractLogger - public static int INFO No 1; public static int DEBUG No 2; public static int ERROR No 3; Protected int level; The next element in the chain or responsibility is protected by AbstractLogger nextLogger; Public void set by NextLogger (AbstractLogger nextLogger nextLogger) - this.nextLogger - nextLogger; - Public invalid logMessage (int level, String message) if (this level of zlt; level) write (message); Step 2 Create specific classes that expand the logger. ConsoleLogger.java public class ConsoleLogger expands AbstractLogger - public consoleLogger (level int) - this.level - level; - @Override protected void, writes (String message) - System.out.println (Standard console:::Logger: - message); ErrorLogger.java public class ErrorLogger expands AbstractLogger - public ErrorLogger (int level) - this.level - level; - @Override protected void write (String message) - System.out.println (File error::::Logger: FileLogger.java public class FileLogger expands AbstractLogger - public FileLogger (int level) - this.level - level; - @Override protected void to write (String message) - System.out.println (File:::Logger: Step 3 Create different types of loggers. ChainPatternDemo.java public class ChainPatternDemo - private static AbstractLogger getChainOfLoggers ( ) . AbstractLogger fileLogger - new FileLogger (AbstractLogger.DEBUG); AbstractLogger consoleLogger - the new ConsoleLogger (AbstractLogger.INFO); errorLogger.setNextLogger (fileLogger); (consoleloger); Return of the Logger error; - Public static emptiness of the main (String) (String) - AbstractLogger loggerChain - getChainOfLoggers (); loggerChain.logMessage (AbstractLogger.INFO, This is information.); loggerChain.logMessage (AbstractLogger.DEBUG, This is the level of debugging.); loggerChain.logMessage Step 4 Withdrawal Check. Standard console::Logger: This is information. File::Logger: This is the level of debugging. Standard console::Logger: This is debugging level information. Error console::Logger: This is error information. File::Logger: This is error information. Standard console::Logger: This is error information. The Command Pattern Command pattern is a data-driven design pattern and falls into the behavioral pattern category. The request is wrapped as a command and sent to the call object. Invoker is looking for a suitable object that can handle that command and send the command to the appropriate object, and that object commands. Implementation We have created an order interface that acts like a command. We've created a Stock class that acts as a query. We have specific command classes BuyStock and SellStock implementation of the order interface that will do the actual processing of commands. A Broker class is created that acts as a call object. He can take orders and order. The Broker object uses a command pattern to determine which object the command will perform, depending on the type of command. CommandPatternDemo, our demo class will use the Broker class to demonstrate the team template. Step 1 Create a command interface. Order.java Public Interface Order - Invalid execution (); Step 2 Create a query class. Stock.java public class Stock - private string name - ABC; Private int number - 10; Public Void to Buy () System.out.println (Stock - Name: Name), Number: bought) ; Step 3 Create specific classes by implementing the Order interface. BuyStock.java public class BuyStock implements the order - private stock abcStock; Public BuyStock (Stock abcStock) - this.abcStock - abcStock; SellStock.java state class SellStock implements the order - private stock abcStock; Public SellStock (Stock abcStock) - this.abcStock - abcStock; Step 4 Create a team call class. Broker.java import java.util.ArrayList; import java.util.List; Public Class Broker - private orderList list - new ArrayList Public void Step 5 Use the Broker class to execute commands. CommandPatternDemo.java public class CommandPatternDemo - publicStocks BuyStock buyStockOrder - new BuyStock (abcStock); SellStock SellStockOrder - new SellStock (abcStock); Broker and new broker broker.takeOrder (buyStockOrder); broker.takeOrder (sellStockOrder); broker.placeOrders(); Step 6 Check output. Stock - Title: ABC, Number: 10 - Bought Shares - Title: ABC, Number: 10 - Sold Pattern Translator Pattern provides a way to evaluate language grammar or expression. This type of pattern fit the behavioral patterns. This pattern includes the implementation of an expression interface that says to interpret a certain context. This pattern is used in character parsing, character processing, etc. A TerminalExpression class is defined that acts as the primary context interpreter in question. Other OrExpression and AndExpression classes are used to create combined expressions. InterpreterPatternDemo, our demo class will use the expression class to create rules and demonstrate expression parsing. Step 1 Create an expression interface. Expression. Java Public Interface Expression - Public Drilling Interpretation (context of the line); Step 2 Create specific implementation classes above the interface. TerminalExpression.java public class TerminalExpression implements the expression @Override - private thong data; Terminal PublicExpression (String Data) this.data - data; OrExpression.java public class OrExpression implements Expression - private expression expr1 - null; private expression expr2 - null; public OrExpression (Expression expr1, expression expr2) @Override - this.expr1 - expr1; this.expr2 - expr2; AndExpression.java of public class AndExpression implements Expression - private expression expr1 - zero; private expression expr2 - null; Public AndExpression (Expression expr1, Expression expr2) @Override - this.expr1 - expr1; this.expr2 - expr2; Step 3 InterpreterPatternDemo uses the expression class to create rules and then analyzes them. InterpreterPatternDemo.java public class InterpreterPatternDemo - /Rule: Robert and John are male public static expressions getMaleExpression () Expression robert - the new TerminalExpression (Robert); Expression john - new TerminalExpression (John); the return of the new OrExpression (Robert, John); Rule: Julie is a married woman, public static expression getMarriedWomanExpression () Expression julie - the new TerminalExpression (Julie); Expression is Married and a New Marriage the return of the new AndExpression (Julie, married); - public static emptiness args) - Expression isMale - getMaleExpression (); Expression Married Female - getMarriedWomanExpression System.out.println (John - man? - isMale.interpret (John); System.out.println (Julie married woman? Married woman. Step 4 Check the output. John is a man? a true Julie married woman? The true Iterator Pattern Iterator pattern is very often used by the design pattern in the Java and .Net programming environments. This template is used to consistently access the items of the collection object without having to know its basic view. The Iterator pattern falls under the category of behavioral models. Implementation We are going to create an Iterator interface that tells the story of the navigation method and the container interface that reruates the iterator. Specific container interface implementation classes will be responsible for implementing the Iterator interface and using it IteratorPatternDemo, our demo class will use NamesRepository, a specific class implementation for printing names stored in the NamesRepository collection. Step 1 Create interfaces. Iterator.java The public object is next Container. Java Public Interface Container - Public Iterator getIterator (); Step 2 Create a specific class by implementing the container interface. This class has an internal Class NameIterator implementation interface Iterator. NameRepository.java public class NameRepository implements Container (public line names) - Robert, John Julie, Laura; @Override public Iterator getIterator - the return of the new NameIterator (); NameIterator's private class implements the Iterator Index; @Override public boolean hasNext () @Override - if (index qlt; names.length) returns correctly; Step 3 Use NameRepository to get the iterator and print out the names. IteratorPatternDemo.java public class IteratorPatternDemo - public static emptiness core (String) - NameRepository namesRepository - new nameRepository (); for (Iterator iter) - namesRepository.getIterator (); iter.hasNext ();)) System.out.println (Name : - name); Step 4 Check the exit. Name : Robert's Name : John Name : Julie Name : Laura Broker Pattern Broker pattern pattern pattern is used to reduce the complexity of communication between multiple objects or classes. This template provides a class of intermediaries that typically handles all communications between different classes and supports easy code support through free connection. The intermediary model falls into the category of behavioral models. Implementation We demonstrate an intermediary template using the example of a chat room where multiple users can send messages to the chat room, and the chat room is required to show messages to all users. We've created ChatRoom and user classes. The user's objects will use the ChatRoom method for messaging. MediatorPatternDemo, our demo class will use the user's objects to show the connection between them. Step 1 Create a class of intermediaries. ChatRoom.java import java.util.Date; ChatRoom Public Class - Public Static Void ShowMessage (User, String Message) System.out.println (new date ().toString() Step 2 Create user class User.java public class User - private line name; the public line getName () - the name of the return; - public invalid setName (String name) - this.name - name; - public user (String name) - this.name - name; - public void sendMessage (String message) ChatRoom.showMessage (this, message) Step 3 Use the user object to show the connection between them. MediatorPatternDemo.java public class MediatorPatternDemo - public static void core (String) - User Robert and new user (Robert); User John and a new user (John); robert.sendMessage (Hello! John!); john.sendMessage (Hello! Robert!); Step 4 Check the exit. Thu Jan 31 16:05:46 IST 2013 (Robert) : Hello! John! Thu Jan 31 16:05:46 IST 2013 (John) : Hello! Robert! The Memento Pattern Memento template is used to reduce the status of an object in the previous state. The Memento pattern falls under the category of behavioral patterns. The Memento Implementation Pattern uses three classes of actors. Memento contains the state of the object that needs to be restored. The creator creates and stores states in Memento and Caretaker objects, which are responsible for restoring the object's condition from Memento. We created Memento, Originator and CareTaker classes. MementoPatternDemo, our demo class will use CareTaker and Originator to demonstrate object recovery. Step 1 Create a Memento class. Memento.java public class Memento - private string state; Public State Memento (String State) - this.state - state; - Public String getState ()) - return state; Step 2 Create Originator Class Originator.java Public Class Originator - Private Row State; Public void setState (String state) this.state - state; - Public String getState () return state; - Public Memento saveStateToMemento () the return of the new Memento (state); Step 3 Create CareTaker-class CareTaker.java import java.util.ArrayList; import java.util.List; CareTaker Public Class - private mementoList list - new ArrayList Public emptiness to add mementoList.add (Memento state); Step 4 Use CareTaker and Originator objects. MementoPatternDemo.java public class MementoPatternDemo - public static emptiness core (String) - - новый создатель (); CareTaker careTaker - новый CareTaker (); создатель.setState (Государственная #1); создатель.setState (Государственная #2); </Memento></Memento> </Memento></Memento> creator.set #3 State careTaker.add (originator.saveStateToMemento()); creator.set #4 State System.out.println creator.getStateFromMemento (careTaker.get(0)); System.out.println (First saved state: - originator.getState ()); creator.getStateFromMemento (careTaker.get(1)) System.out.println (Second Saved State: - originator.getState ()); Step 5 Check the exit. Current State: State #4 First Saved State: State #2 Second Saved State: The Observer pattern #3 used when there is one of many relationships between objects, for example, if one object is changed, its depenedent objects must be notified automatically. The observer pattern falls under the category of behavioral patterns. The Implementation Observer template uses three classes of actors. Theme, observer and client. The subject, the object, having methods of attaching and de-attache observers to the client's object. We created Subject, Observer abstract class and specific classes that expand the Observer abstract class. ObserverPatternDemo, our demo class will use objects of objects and specific classes to show the observer pattern in action. Step 1 Create a subject class. Subject.java import java.util.ArrayList; import java.util.List; Public Class Theme - Private Observer List - New ArrayList Private int state; Public int getState - return state; - Public invalid set of State (int state) - this.state - state; notifyAllObservers (); - Public invalid attache Step 2 Create an observer class. Observer.java public abstract class Observer - protected subject; Public abstract invalid update Step 3 Create specific classes of observers BinaryObserver.java public class BinaryObserver expands Observer @Override' public BinaryObserver (Subject of subject) this.subject this.subject.attach OctalObserver.java public class OctalObserver expands Observer' public OctalObserver (Subject of subject) this.subject this.subject.attach - @Override public invalid update () - System.out.println, Octal String: HexaObserver.java public class HexaObserver expands Public HexaObserver (Subject subject) this.subject.attach - @Override public invalid update () - System.out.println, Hex String: Step 4 Using Theme and Specific Observer Objects. ObserverPatternDemo.java public class ObserverPatternDemo - public static emptiness core (String) - Theme - a new subject (); new HexaObserver (subject); new/Observer New BinaryObserver System.out.println (First state change: 15); subject.setState (15); System.out.println subject.setState (10); Step 5 Check the output. First status change: 15 Hex String: F Octal String: 17 Binary String: 1111 Second status change: 10 Hex String: A Octal String: 12 Binary String: 1010 State Pattern In State pattern class behavior change depending on its condition. This type of design pattern approached the behavior pattern. In the state pattern, we create objects representing different states and a contextual object whose behavior changes as the state object changes. Implementation We are going to create a state interface that determines the action and specific state classes of implementation of the state interface. Context is a class that the state carries. StaePatternDemo, our demo class will use context and state objects to demonstrate changes in Context behavior depending on the type of state it is in. Step 1 Create an interface. Image.java Public Interface State - Public Void doAction (ContextUal Context); Step 2 Create specific classes by implementing the same interface. StartState.java public class StartState implements state - public void doAction (context context) - System.out.println (Player is in a state of launch); context.setState StopState.java public class StopState implements state - public void doAction (context context) - System.out.println (Player is in a state of stoppage); context.setState Step 3 Create a class context. Context.java State Class Context - Private State; Public context () state of Yu null; - public set of emptinessState (state) this.state - state; Step 4 Use context to see changes in behavior as state changes. StatePatternDemo.java public class StatePatternDemo - public static emptiness core (String) - Context - new context (); StartState StartState - a new beginningsstate (); startState.doAction System.out.println (context.getState().toString()); StopState stopState - a new stop-state (); stopState.doAction System.out.println (context.getState().toString()); Step 5 Check the output. The player is in the Start State Player's state of start is in the Null Object Pattern In Null Object stop state, and the zero object replaces the NULL instance check. Instead of putting, if you check for zero value, Null Object reflects nothing to do the relationship. This Null can also be used to ensure default behavior if the data is not available. In the Null Object template, we create class that identifies the various operations that need to be performed, create classes that expand this class, and a null object class that doesn't provide anything for the implementation of that class, and will be used, seemingly, where we need to check zero value. The implementation we're going to AbstractCustomer abstract class, which defines the disgrace, here is the customer's name and specific classes that expand the AbstractCustomer class. The CustomerFactory factory class is created to return RealCustomer or NullCustomer objects based on the customer's name given to them. NullPatternDemo, our demo class will use CustomerFactory to demonstrate the use of the Null Object template. Step 1 Create an abstract class. AbstractCustomer.java public abstract class AbstractCustomer - protected string name; Public abstract boulean isNil getName public abstract line Step 2 Create specific classes that expand the above class. RealCustomer.java public class RealCustomer expands AbstractCustomer - public RealCustomer (String name) - @Override this.name @Override - name; NullCustomer.java public class NullCustomer expands AbstractCustomer - @Override the public line getName () @Override - the return is not available in the customer database; Step 3 Create a CustomerFactory Class. CustomerFactory.java public class CustomerFactory - public static final lines - Rob, Joe, Julie; Public Static AbstractCustomer getCustomer (String name) for (int i y 0; i zlt; names.length; i)) Step 4 Use CustomerFactory to get either RealCustomer or NullCustomer objects based on the customer's name passed to it. NullPatternDemo.java public class NullPatternDemo - public static emptiness core (String) - AbstractCustomer customer1 - CustomerFactory.getCustomer (Rob); AbstractCustomer - CustomerFactory.getCustomer AbstractCustomer - CustomerFactory.getCustomer AbstractCustomer - CustomerFactory.getCustomer System.out.println System.out.println (customer1.getName();; System.out.println (customer2.getName();; System.out.println (customer3.getName.);; System.out.println (customer4.getName()); Step 5 Check the output. Rob Not Available customers in the Julie Not Available in Customer Database Strategy Strategy In Strategy Strategy, class behavior or its algorithm can be changed during operation time. This type of design pattern approached the behavior pattern. In the Strategy template, we create objects that represent different strategies and a contextual object whose behavior varies depending on the object of the strategy. The strategy object changes the context object execution algorithm. Implementation We are going to create a Strategy interface that defines actions and specific strategy classes that implement the Strategy interface. Context is the class in which strategy is used. StrategyPatternDemo, our demo class will use contextual and strategic objects to demonstrate changes in Context behavior about the strategy he deploys or uses. Step 1 Create an interface. Strategy. Java Public Interface Strategy - public int doOperation (int num1, int num2); Step 2 Create specific classes by implementing the same interface. OperationAdd implements a strategy @Override public int doOperation (int num1, int num2) Operation Substract.java of the public class OperationSubstract implements a strategy @Override public int dorationOperation (int num1, int num2) - return num1 - num2; Operation Multiple Java @Override OperationMultipli implements strategy and public collaboration (int num1, int num2) Step 3 Create a class context. Context.java Public Class Context is a private strategy strategy; Public Context (Strategy Strategy) this.strategy - Public int executeStrategy (int num1, int num2) Step 4 Use context to see changes in behavior when it changes its Strategy. StatePatternDemo.java public class StrategyPatternDemo - Public Static Void Core (String)) - Context Context - New Context (New OperationAdd)); System.out.println (10 and 5 - context.executeStrategy (10, 5)); Context - New Context (new Operation Subject;); System.out.println (10 - 5 - context.executeStrategy (10, 5)); Context - a new context (new operationMultipli System.out.println (10 and 5 - context.executeStrategy (10, 5)); Step 5 Check the exit. 10 - 5 - 15 10 - 5 - 5 10 and 50 Pattern Pattern pattern in the template template, abstract class provides a certain path (s)/pattern (s) to perform their methods. Its subclasses can override the implementation of the method according to the necessary basis, but the call must be in the same way as defined by the abstract class. This pattern has reached the category of behavior patterns. Implementation We're going to create an abstract game class that defines operations with a template method that will be final, so that it can't be redefined. Cricket and football are specific classes to extend the game and override its methods. TemplatePatternDemo, our demo class will use the game to demonstrate the use of the template template. Step 1 Create an abstract class with the final pattern method. Game.java public abstract class Game - abstract emptiness is initiated (); Abstract invalid startPlay abstract invalid endPlay patterned method of public final invalidation game ())/initiate the game to initiate (); start game startPlay (); end game endPlay (); Step 2 Create specific classes that expand the above class. Cricket public class Cricket.java expands the game no @Override void endPlay () - System.out.println (Cricket game over!) @Override; Start playing.); @Override startPlay () - System.out.println (Cricket game has begun. game!); Football.java public-class Football expands the game - @Override invalid invalid System.out.println (Football game over!); @Override to initialize () - System.out.println (Football game initiated! @Override startPlay () - System.out.println (Football match has begun. Step 3 Use the game pattern method of playing the game () to demonstrate a certain way of playing. TemplatePatternDemo.java public class TemplatePatternDemo - public static emptiness core (String) - Playing new cricket (); game.play (); System.out.println(); Game - new football game.play(); Step 4 Check the output. Cricket game initiated! Start playing. The game of cricket has begun. Enjoy the game! Cricket game over! The football game is initiated! Start playing. The football match began. Enjoy the game! The football match is over! The visitor pattern in the visitor pattern, we use a class of visitors that changes the algorithm of the performance of the class element. Thus, the algorithm of the item's execution can change as many visitors. This pattern has reached the category of behavior patterns. In accordance with the template, the item object must accept the visitor's object so that the visitor's object is charring the operation on the item object. Implementation We are going to create a ComputerPart interface that defines the reception of opearation. The keyboard, mouse, monitor and computer are specific classes of ComputerPart interface implementation. We'll identify a different ComputerPartVisitor interface that will identify visitor class operations. The computer uses a particular visitor for appropriate action. VisitorPatternDemo, our demo class will use computer, ComputerPartVisitor classes to demonstrate the use of the visitor template. Step 1 Identify the interface to represent the item. ComputerPart.java public interface of computerPart class - public invalid reception (ComputerPartVisitor computerPartVisitor); Step 2 Create specific classes that expand the above class. Keyboard keyboard keyboard of public class Keyboard @Override ComputerPart - public invalid reception (ComputerPartVisitor computerPartVisitor) - computerPartVisitor.visit (this); Monitor Monitor Monitor implements ComputerPart @Override a public invalid (ComputerPartVisitor computerPartVisitor) - computerPartVisitor.visit (it); Mouse.java Public Class Mouse implements ComputerPart - @Override public acceptance of void (ComputerPartVisitor computerPartVisitor) - computerPartVisitor.visit (it); Computer.java's public-class computer sells computer Parts and ComputerPart; Public computer ()) parts - new ComputerPart, new keyboard ,), new monitor (); - @Override public invalid (ComputerPartVisitor computerPartVisitor) - for (int i q 0; i q lt; parts.length; i'i'.accept Step 3 Identify the interface for the visitor view. ComputerPartVisitor.java public interface ComputerPartVisitor - public invalid visit (Computer computer); Public blank visit (Mouse mouse); Mice); invalid visit (keyboard keyboard); Public blank visit (monitor-monitor); Step 4 Create a specific visitor by implementing the above class. ComputerPartDisplayVisitor.java public-class ComputerPartDisplayVisitor implements ComputerPartVisitor - @Override public invalid visit (Computer Computer) - System.out.println (Computer Display.); - @Override public invalid visit (mouse) - System.out.println (Mouse Display.); - @Override public invalid visit (keyboard keyboard) @Override - System.out.println (Keyboard display.); Step 5 Use ComputerPartDisplayVisitor to display computer parts. VisitorPatternDemo.java public class VisitorPatternDemo - public static emptiness core (String) - ComputerPart computer - new computer (); computer.accept Step 6 Check the exit. Showing the mouse. Displaying the keyboard. Displaying the monitor. Showing your computer. The MVC Pattern MVC pattern means model-view-controller pattern. This template is used to share the app's problems. Model - The model represents an object or JAVA POJO that carries data. It can also have the logic to update the controller if its data changes. View - The view represents a visualization of the data that the model contains. Controller - Controller acts on both the model and the view. It controls the flow of data to the model object and updates the view whenever the data changes. It keeps the look and model separate. Implementation We are going to create a student facility that acts as a model. StudentView will be a viewing class that can print student data on the console and StudentController is the class controller responsible for storing data in the student facility and updating the StudentView view accordingly. MVCPatternDemo, our demo class will use StudentController to demonstrate the use of the MVC template. Step 1 Create a model. Student. Java Public Class Student - Private String RollNo; The private name of the string; getRollNo public line - reverse rollNo; - public invalid setRollNo (String rollNo) this.name - this.rollNo - rollNo; Step 2 Create a view. StudentView.java public class StudentView - public void printStudentDetails (String studentName, String studentRollNo) System.out.println (Student:); System.out.println System.out.println (Roll No: and student rollNo); Step 3 Create a controller. StudentController.java StudentController is a private student model; Private StudentView public StudentController (Student Model, StudentView) - this.model - model; this.view - view; Public invalid (String name)' model.setName (имя); - публичная строка getStudentName()) getStudentName()) - Public Invalid updateView ()) view.printStudentDetails (model.getName(), model.getRollNo(); Step 4 Use StudentController methods to demonstrate the use of the MVC design pattern. MVCPatternDemo.java public class MVCPatternDemo - public static void core (String) - /fetch student record based on its roll is not from the database Student model - retriveStudentFromDatabase (); Create a view: write student details on StudentView view - the new StudentView StudentController Controller - new StudentController (model, view); controller.updateView (); Update the data controller model. controller.updateView (); - Private static student retriveStudentFromDatabase ()) Student-student - new student (); student.setName (Robert); student.setRollNo (10); The return of the student; Step 5 Check the exit. Student: Name: Robert Roll No: 10 Student: Name: Julie Roll No: 10 Business Delegate Template Business Delegate pattern is used to disengage the presentation level and business level. This is mainly used to reduce communication or remote access to business-level code in view-level code. At the business level, we follow organizations. Customer - The presentation level code can be JSP, servlet, or java UI code. A business delegate is a single class of entry points for client organizations to ensure access to business services. LookUp Service - Lookup is responsible for the relative implementation of the business and the provision of business objects access to the business delegate's facility. Business service - Business service interface. Specific classes implement this business service to ensure the actual logic of business implementation. Implementation We are going to create a client, BusinessDelegate, BusinessService, LookUpService, JMSService and EJBService, representing different organizations of the business delegate template. BusinessDelegatePatternDemo, our

demo class will use BusinessDelegate and the customer to demonstrate the use of the Business Delegate template. Step 1 Create a BusinessService interface. BusinessService.java public interface BusinessService - public void doProcessing (); Step 2 Create Concreate service classes. EJBService.java public class EJBService implements BusinessService - @Override public space doProcessing () - System.out.println (Problem Processing by calling EJB); JMSService.java public class JMSService implements BusinessService - @Override public space doProcessing () - System.out.println (Task Processing by Calling JMS); Step 3 Create a business service for inspection. BusinessLookUp.java public class BusinessLookUp - Public BusinessService getBusinessService (String serviceType) if (serviceType.equalsIgnoreCase (EJB)) is still Return of the new JMSService Step 4 Create a Business Delegate. BusinessLookUp.java public class BusinessDelegate - private BusinessLookUp lookupService - new BusinessLookUp (); Private private BusinessService; Private String ServiceType; Public Void SetServiceType (String serviceType) - this.serviceType - serviceType; - Public void doTask ()) businessService - lookupService.getBusinessService (serviceType); businessService.doProcessing (); Step 5 Create a customer. Student.java Public Class Customer - BusinessDelegate BusinessService; BusinessDelegate businessService - this.businessService - businessService; Step 6 Use BusinessDelegate and Client classes to demonstrate the Business Delegate template. BusinessDelegatePatternDemo.java public class BusinessDelegatePatternDemo - public static emptiness of the core (String'args) - BusinessDelegate businessDelegate - the new BusinessDelegate (); businessDelegate.setServiceType (EJB); Customer and new client (businessDelegate); client.doTask(); businessDelegate.setServiceType (JMS); client.doTask(); Step 7 Check the output. The processing task, citing the EJB processing task, referring to the JMS Service Composite Entity Entity Entity Entity template, is used in the EJB preservation mechanism. The composite entity is an EJB-bean that represents a graph of objects. When the composite entity is updated, internally dependent bean objects are automatically updated, as the EJB is controlled by the bean essence. Below are the participants in Composite Entity Bean. Composite essence - This is the main essence of beans. It can be coarse-grained or may contain a large-grain object that will be used for conservation purposes. Rough grain object -This object contains dependent objects. It has its own life cycle and also manages the life cycle of dependent objects. Dependent object - Dependent objects are an object that depends on a coarse-grain object for its life cycle. Strategies - Strategies are a way to implement a composite entity. Implementation We are going to create a CompositeEntity object that acts as CompositeEntity. CoarseGrainedObject is a class that contains dependent objects. CompositeEntityPatternDemo, our demo class will use the customer class to demonstrate the use of the Composite Entity template. Step 1 Create dependent objects. DependentObject1.java public class DependentObject1 - personal string data; Public set of invalid data data - this.data DependentObject2.java public class DependentObject2 - personal string data; public invalid setData - this.data Step 2 Create a coarse-grained object. CoarseGrainedObject.java DependentObject2 do2 - new dependentObject2(); public void setData (String data1, String Data2) do1.setData (data1); do2.setData (data2); Step 3 Create Essence. Essence. публичный класс CompositeEntity - частный CoarseGrainedObject cgo - новый CoarseGrainedObject (); публичная пустота setData (String data1, String data2) cgo.setData (data1, data2); Шаг 4 Создайте класс клиента для использования Composite Entity. Client.java public class Client - частная композитность CompositeEntity - новая CompositeEntity (); публичная пустота printData ()) для (int i й 0; i < compositeentity.getdata().length;= i++)= {= system.out.println(data:= = += compositeentity.getdata()[i]);= }= }= public= void= setdata(string= data1,= string= data2){= compositeentity.setdata(data1,= data2);= }= }= step= 5= use= the= client= to= demonstrate= composite= entity= design= pattern= usage.= compositeentitypatterndemo.java= public= class= compositeentitypatterndemo= {= public= static= void= main(string[]= args)= {= client= client=new client();= client.setdata(test,= data);= client.printdata();= client.setdata(second= test,= data1);= client.printdata();= }= }= step= 6= verify= the= output.= data:= test= data:= data= data:= second= test= data:= data1= data= access= object= pattern= data= access= object= pattern= or= dao= pattern= is= used= to= separate= low= level= data= accessing= api= or= operations= from= high= level= business= services.= following= are= the= participants= in= data= access= object= pattern.= data= access= object= interface= -= this= interface= defines= the= standard= operations= to= be= performed= on= a= model= object(s).= data= access= object= concrete= class= -this= class= implements= above= interface.= this= class= is= responsible= to= get= data= from= a= datasource= which= can= be= database= xml= or= any= other= storage= mechanism.= model= object= or= value= object= -= this= object= is= simple= pojo= containing= get/set= methods= to= store= data= retrieved= using= dao= class.= implementation= we're= going= to= create= a= student= object= acting= as= a= model= or= value= object.studentdao= is= data= access= object= interface.studentdaoimpl= is= concrete= class= implementing= data= access= object= interface.= daopatterndemo,= our= demo= class= will= use= studentdao= demonstrate= use= of= data= access= object= pattern.= step= 1= create= value= object.= student.java= public= class= student= {= private= string= name;= private= int= rollno;= student(string= name,= int= rollno){= this.name=name; this.rollno=rollNo; } = public= string= getname()= {= return= name;= }= public= void= setname(string= name)= {= this.name=name; } = public= int= getrollno()= {= return= rollno;= }= public= void= setrollno(int= rollno)= {= this.rollno=rollNo; } = }= step= 2= create= data= access= object= interface.= studentdao.java= import= java.util.list;= public= interface= studentdao= {= public=> <Student>getAllStudents(); общественный Студенческий getStudent (int rollNo); публичное недействительное обновлениеStudent (Студент-студент); публичная (Student-student); Step 3 Create a concreate class by implementing the above interface. StudentDaoImpl.java import java.util.ArrayList; import java.util.List; StudentDaoImpl's community class implements StudentDao //list works as a database of students'; StudentDaoImpl () - students - the new ArrayList Student1 - new student (Robert, 0); Student2 - new student (John, 1); students.add students.add - @Override public cancellation of deleteStudent (Student) - students.remove (student.getRollNo); System.out.println (Student: Roll No - student.getRollNo () removed from the database); // //retrive list of students from the database @Override a public list of students (getAllStudents) - return of students; - @Override public student getStudent (int rollNo) @Override - return students.get (rollNo); System.out.println (Student: Roll No - student.getRollNo , updated in the database); Step 4 Use StudentDao to demonstrate the use of the data access object template. CompositeEntityPatternDemo.java public class DaoPatternDemo - Public Static Void Core (String) - StudentDao studentDao - new StudentDaoImpl (); Print all students for (Student: studentDao.getAllStudents)- System.out.println (Student: RollNo : s'student.getRollNo(), Name: 'student.getName,'; studentDao.updateStudent (student); Get student studentDao.getStudent (0); System.out.println (Student: RollNo : s'student.getRollNo),' Name: 'student.getName(' Step 5 Conclusion Check. Student: RollNo: 0, Name : Robert Student: RollNo: 1, Name: John Student: Roll No 0, updated in Student database: RollNo: 0, Name: Michael The front controller pattern Design pattern is used to provide a centralized request processing mechanism, so that all requests will be processed by one handler. Front Controller - Single handler for all kinds of requests coming into the app (based on web/desktop). Dispatcher - The front controller can use the dispatcher's object, which can send the request to the appropriate particular handler. View - Views are the object for which requests are made. Implementation We are going to create FrontController, Dispatcher to act as front controller and dispatcher respectively. HomeView and StudentView present a variety of views for which queries can come to the front controller. FrontControllerPatternDemo, our demo class will use FrontController Ato to demonstrate the front controller design template. Step 1 Create views. HomeView.java public-class HomeView - Public Invalid show ()) System.out.println StudentView.java public class StudentView - Public Invalid Show ()) System.out.println </Student> </Student> Step 2 Create a dispatcher. Dispatcher.java Public Class Dispatch - Private StudentView studentView; Private HomeView homeView; Public Manager - StudentView - new StudentView homeView - new HomeView - Public invalid dispatch (request line) if (request.equalsIgnoreCase (STUDENT) else homeView.show (); Step 3 Create FrontController Context.java State-class FrontController - private dispatcher; public FrontController (); dispatcher - new dispatcher (); - private boolean ishenticUser ()) - System.out. The return is true. Public void of sendingReest (String request) /journal of each trackRequest request (request); User authentication if (isAuthenticUser()Step 4 Use FrontController to demonstrate the design pattern of the front controller. FrontControllerPatternDemo.java public class FrontControllerPatternDemo - public static void core (String) - FrontController frontController - new FrontController(); frontController.dispatchRequest (HOME); frontController.dispatchRequest Step 5 Check the exit. Requested page: HOME USER is successfully authentic. Display of the Page home page requested: STUDENT USER has been successfully verified. Displaying the Student Page Interception Pattern Pattern Is Used when we want to do some pre-processing/post processing with the request or response of the application. Filters are defined and applied on request before submitting the request to the actual target application. Filters can authenticate/authorize/register or track a request and then refer requests to the appropriate handlers. Below are the entities of this type of design pattern. Filter - Filter that will perform a specific task before or after the request is made by the query handler. Filter Chain - The filter chain carries multiple filters and help run them in a certain order on target. Target - Target - Filter Manager Query Handler - Filter Manager manages filters and filter chain. Customer - Customer is the object that sends a request to the target. Implementation We are going to create FilterChain, FilterManager, Target, Customer as various objects representing our faces. AuthenticationFilter and DebugFilter are concrete filters. InterceptingFilterDemo, our demo class will use the customer to demonstrate the interception of the filter design pattern. Step 1 Create a filter interface. Filter.java public interface filter - public execution of emptiness (line request); Step 2 Create concrete filters. AuthenticationFilter.java public class AuthenticationFilter implements filter - public invalid (Line request) System.out.println Request); DebugFilter's public class implements Filter - Public Invalid (Line Request) System.out.println (request log: - request); Step 3 Create a target. Java public class Target - Public Invalidity (Line Request) System.out.println (Request: - Request); Step 4 Create a filter chain FilterChain.java import java.util.ArrayList; import java.util.List; FilterChain public class - private list filters - new ArrayList A private target Public void filters.add - Public invalid (Line Request) for (Filter filter: filters) - filter.execute (request); Step 5 Create FilterManager.java filter manager FilterManager - FilterChain filterChain; Public FilterManager (target) filterChain - the new FilterChain (); filterChain.setTarget (target); Step 6 Create Client.java Customer Customer - FilterManager filterManager; FilterManager (FilterManager filterManager) - this.filterManager - filterManager; - public void sendRequest (Request line) filterManager.filterRequest (request); Step 7 Use the customer to demonstrate the design pattern of intercepting filters. FrontControllerPatternDemo.java public class InterceptingFilterDemo - public static emptiness core (String) - FilterManager filterManager - new FilterManager (new target)); filterManager.setFilter filterManager.setFilter (new DebugFilter) Client-client and new client (); client.setFilterManager client.sendRequest Step 8 Check the output. Authentication Request: HOME Request Log: HOME Execution Request: HOME Service Locator The Locator Pattern Locator Service Template is used when we want to find different services using JNDI Search. Given the high cost of searching for JNDI for the service, the Service Locator template uses caching technique. For the first time, Service Locator is looking for a service in JNDI and cachet the service object. Further search or the same service through Service Locator is done in the cache, which greatly improves the performance of the application. Below are the entities of this type of design pattern. Service - Actual service that handled the request. The link to such a service should be reviewed on the JNDI server. Context/Original Context -JNDI Context, carries a link to the service used for verification purposes. Locator service - Service Locator - is a single point of contact for the view of JNDI search services, caching services. Cash - Cache to store links to services to reuse them Client - Customer is an object that calls services through ServiceLocator. </Filter></Filter>We will create ServiceLocator, InitialContext, Cache, Service as various objects representing our essences. Service1 and Service2 provide specific services. ServiceLocatorPatternDemo, our demo class acts as a client here and will use ServiceLocator to demonstrate the service locator design pattern. Step 1 Create a service interface. Service.java Public Interface Service - getName public line Public invalidation Step 2 Creating specific services. Service1.java Public Class Service1 implements service @Override - public void of execution ()) System.out.println (Execution Service1); Service2.java Public Class Service2 implements @Override Service - Public Invalidity ()) System.out.println (Service2); Step 3 Create initialContext to search JNDI InitialContext.java public class InitialContext - search for public objects (String jndiName) if (jndiName.equalsIgnoreCase (SERVICE1)) still if (jndiName.equalsIgnoreCase (SERVICE2)) System.out.println\nSearch and the creation of a new Service2 facility); The return of the new Service2 Step 4 Create cache.java import java.util.ArrayList; import java.util.List; Public-class cash services - private services on the list; Public cache (service) - new ArrayList (); -State service getService (String serviceName) for (Service: services) - if the service.service.getName (.).equalsIgnoreCase (serviceName)) reverse service; Return invalid; - Public emptiness addService (Service newService) boolean exists - false; for (Service: services) - if (service.getName ().equalsIgnoreCase (newService.getName ())) exists - true; Step 5 Create Locator Service.java public-class ServiceLocator - private static cache cache; static cache - new cache - State static service getService (String jndiName) - Service - cache.getService (jndiName); If (service ! Original ContextContext - new initialContext(); Service1 (Service)context.lookup (jndiName); cache.addService (service1); service return1; Step 6 Use ServiceLocator to demonstrate the design pattern of the service locator. ServiceLocatorPatternDemo.java of the state class ServiceLocatorPatternDemo - public static emptiness of the core (String) - Service - ServiceLocator.getService (Service1); service.execute (); Service - ServiceLocator.getService (Service2); service.execute (); Service - ServiceLocator.getService (Service1); service.execute (); Service - ServiceLocator.getService (Service2); service.execute (); Step 7 Check the exit. Search and create a new Service1Searching for and creating a new Service2 facility by performing Service2 Returning cached Service1, performing Service1 Returning cached Service2 object Executing Service2 Transfer Object Pattern The Transfer Object Object Object Is used when we want to transmit data with multiple attributes in one shot from customer to server. The transmission object is also known as the value object. Transfer Object is a simple POJO class with getter/setter and serial methods, so that it can be transmitted over the network. He has no behavior. The Server Side business class usually receives data from the database and fills in the POJO and sends it to the customer or passes by cost. For the customer, the transmission object is only for reading. The client can create their own transfer object and transfer it to the server to update the values in the database in one shot. Below are the entities of this type of design pattern. A business object is a business service that fills the data transfer facility. Object transfer -Just POJO, having methods to dial/get attributes only. Customer - Customer either asks or sends the object of transfer to the business object. Implementation We are going to create StudentBO as a business object, Student as a transfer object representing our individuals. TransferObjectPatternDemo, our demo class acts as a client here and will use StudentBO and students to demonstrate the design pattern of transfer facilities. Step 1 Create a transmission object. StudentVO.java public class StudentVO - private name of the line; Private int rollNo; StudentVO (String name, int rollNo) - this.name - name; this.rollNo - rollNo; - Public String getName - return name; - Public set of voids (String name) - this.name - name; Step 2 Create a business object. StudentBO.java import java.util.ArrayList; import java.util.List; StudentBO Community Class /List works as a student list database; Public StudentBO ()) students - the new ArrayList'lt;StudentVO'gt; Student VO Student1 - New Student (Robert,0); Student VOY Student2 - New Student (John,1); students.add students.add - public void removed by student and students. System.out.println (Student: Roll No - student.getRollNo () removed from the database); / //retrive list of students from the database of public list getAllStudents () - return of students; - StudentVO getStudent (int rollNo) - return students.get (rollNo); - Public invalid updateStudent (student) - students.get (student.getRollNo ())setName (student.getRollNo)) System.out.println (Student: Roll No - student.getRollNo () updated in the database); Step 3 Use StudentBO to demonstrate the design pattern of transmission objects. TransferObjectPatternDemo.java класса TransferObjectPatternDemo - публичная статическая пустота основной (String)</StudentVO> </StudentVO> </StudentVO> </StudentVO> StudentBusiness - the new StudentBO (); PrintAll Students for (Student: studentBusinessObject.getAllStudents)- System.out.println (Student: RollNo: s'student.getRollNo),Name: 'student.getName,'; student.setName (Michael); studentBusinessObject.updateStudent (student); Get Student BusinessObject.getStudent (0); System.out.println (Student: RollNo : s'student.getRollNo(), Name: 'student.getName(' Step 4 Conclusion. Student: RollNo : 0, Name : Robert Student: RollNo: 1, Name: John Student: Roll No 0, updated in Student database: 'RollNo: 0, Name: Michael design patterns tutorial point pdf. design patterns tutorial point c#. structural design patterns tutorial point. behavioral design patterns tutorial point. javascript design patterns tutorial point. java design patterns tutorial point pdf. singleton design patterns tutorial point. design patterns in python tutorial point