**Form Diskusi Pertemuan 12**

**Design Patterns 02**

**Tujuan diskusi:**

* Mahasiswa mampu mengimplementasi Factory Pattern dan Singleton Pattern dengan menggunakan Bahasa Java.

**Tutorial 12.1: Factory Pattern**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1. Factory Pattern merupakan pattern yang digunakan untuk menciptakan sebuah objek dari kelas tertentu. Dalam membuat objek dengan pattern ini, kita tidak dihadapkan dengan lojik bagaimana objek dibuat, melainkan penciptaan objek dilakukan melalui sebuah antarmuka. Struktur pattern dapat dilihat pada gambar 3 berikut ini:   Gambar 1. Struktur Factory Pattern   1. Dengan menggunakan struktur composite pada tutorial 11.4, Pada tutorial ini, akan diimplementasikan Factory Pattern, dengan kelas-kelas yang terlibat seperti pada gambar 2 berikut:   Gambar 2. Struktur Lengkap Factory Pattern Sesuai Kasus   |  |  |  | | --- | --- | --- | | No | Kelas dalam Factory Pattern | Kelas Tutorial | | 1 | Abstract Product | Employee | | 2 | Concrete Product | Manager | | 3 | Concrete Product | Staf | | 4 | Factory | EmployeeFactory | | 5 | Client | FactoryPatternTest |   ----------   1. Buatlah project dengan nama **FactoryPattern** 2. Gunakan kelas-kelas yang ada dalam tutorial 11.4, yaitu : **Employee**, **Manager**, **Staf**, dan **Asuransi**. 3. Buatlah kelas **EmployeeFactory**, dengan kode program berikut ini:   public class EmployeeFactory {  public Employee create(**String type**, String name, String role,  Asuransi asr) {  if (type.equalsIgnoreCase("Manager")) {  System.out.println("Membuat Instance Manager");  **return new Manager(name, role, asr);**  }  else { // Penciptaan objek Staf  System.out.println("Membuat Instance Staf");  **return new Staf(name, role, asr);**  }  }  }   1. Buat kelas **FactoryPatternTest** menjadi berikut ini:   public class FactoryPatternTest {  public static void main(String[] args) throws IOException {  **EmployeeFactory ef = new EmployeeFactory();**  Asuransi kis = new Asuransi();  Employee direktur = **ef.create**("**Manager**", "Arief Nugraha", "Direktur", kis);  Employee mankeu = **ef.create** ("**Manager**", "Suryana", "Manajer Keuangan", kis);  Employee mansdm = **ef.create** ("**Manager**", "Susila","Manajer SDM", kis);  Employee kaakun = **ef.create** ("**Manager**", "Septarini","Kabag Akuntansi", kis);  Employee kaaudt = **ef.create** ("**Manager**", "Herjuna","Kabag Audit", kis);  Employee kapsdm = **ef.create** ("**Manager**", "Dewi Yani", "Kabag SDM", kis);  Employee kaprom = **ef.create** ("**Manager**", "Eleonora", "Kabag Promosi", kis);  Employee skeu1 = **ef.create** ("**Staf**", "Panduwinata", "Staf Keuangan", kis);  Employee skeu2 = **ef.create** ("**Staf**", "Megadewi", "Staf Keuangan", kis);  Employee ssdm1 = **ef.create** ("**Staf**", "Trifina", "Staf SDM", kis);  Employee ssdm2 = **ef.create** ("**Staf**", "Kanda", "Staf SDM", kis);  Employee ssdm3 = **ef.create** ("**Staf**", "Arjuna", "Staf SDM", kis);  direktur.addChild(mankeu); direktur.addChild(mansdm);  mankeu.addChild(kaakun); mankeu.addChild(kaaudt);  kaakun.addChild(skeu1);  kaaudt.addChild(skeu2);  mansdm.addChild(kapsdm); mansdm.addChild(kaprom);  kapsdm.addChild(ssdm1); kapsdm.addChild(ssdm2);  kaprom.addChild(ssdm3);  direktur.traverseEmp();  InputStreamReader values = new InputStreamReader (System.in);  BufferedReader br = new BufferedReader(values);  do {  System.out.print("\nInput kenaikan premi [0..1] : ");  String valueStr = br.readLine();  if (valueStr.equalsIgnoreCase("00"))  break;  kis.setState(Float.parseFloat(valueStr));  direktur.traverseEmp();  } while(true);  }  }   1. Eksekusi program. |
| *Apa yang bisa Anda simpulkan dengan menggunakan Factory Pattern ?*  Tergantung dari |

**Tutorial 12.2: Singleton Pattern**

|  |
| --- |
| 1. Singleton Pattern merupakan pattern yang digunakan untuk menjamin bahwa objek dari sebuah yang diciptakan hanya satu saja.   Gambar 3. Struktur Singleton Pattern   1. Tutorial 12.2 akan mengimplementasikan Singleton Pattern, di mana kelas **Asuransi** akan dibuat menjadi singleton, dalam arti instance kelas ini hanya dibatasi satu saja. 2. Buatlah project dengan nama **SingletonPattern** 3. Gunakan hasil tutorial 12.1, dengan menggunakan semua kelas yang ada, kecuali kelas **FactoryPatternTest.** 4. Lakukan modifikasi terhadap kelas **Asuransi**, dengan kode program berikut ini. Kode ini menunjukkan objek kelas **Asuransi** hanya dapat diciptakan satu saja. (Catatan: kode program berwarna merah menunjukkan implementasi singleton)   import java.util.Vector;  public class Asuransi {  ///////////////////////////////////////////////////////  // atribut dan fungsi kelas Asuransi dalam implementasi  // pattern singleton  private static Asuransi asuransi = null;  public static Asuransi getInstance() {  if(asuransi == null) {  asuransi = new Asuransi();  System.out.println("Instansi kelas Asuransi dibuat");  }  else {  System.out.println("Instansi kelas Asuransi sudah ada");  }  return asuransi;  }  //////////////////////////////////////////////////////  // atribut kelas Asuransi sebagai subject dalam pattern observer  private Vector<Employee> views;  ////////////////////////////////////////////////////////////////  // fungsi konstruktor dibuat private sebagai implementasi pattern  // singleton  private Asuransi() {  views = new Vector<Employee>();  }  ////////////////////////////////////////////////////////////////  // fungsi-fungsi kelas Asuransi sebagai subject dalam  // pattern observer  public void attach (Employee obs) {  views.add(obs);  }  public void notifyObserver(float val) {  for (int i=0; i < views.size(); i++)  views.elementAt(i).update(val);  }  public void setState(float val) {  notifyObserver(val);  }  }   1. Buatlah kelas **SingletonPatternTest** dengan kode berikut ini:   public class SingletonPatternTest {  public static void main(String[] args) throws IOException {  EmployeeFactory ef = new EmployeeFactory();  **Asuransi kis = Asuransi.getInstance();**  **Asuransi kis2 = Asuransi.getInstance();**  Employee direktur = ef.create("Manager", "Arief Nugraha", "Direktur", kis);  Employee mankeu = ef.create ("Manager", "Suryana", "Manajer Keuangan", kis);  Employee mansdm = ef.create ("Manager", "Susila","Manajer SDM", kis);  Employee kaakun = ef.create ("Manager", "Septarini","Kabag Akuntansi", kis);  Employee kaaudt = ef.create ("Manager", "Herjuna","Kabag Audit", kis);  Employee kapsdm = ef.create ("Manager", "Dewi Yani", "Kabag SDM", kis);  Employee kaprom = ef.create ("Manager", "Eleonora", "Kabag Promosi", kis);  Employee skeu1 = ef.create ("Staf", "Panduwinata", "Staf Keuangan", kis);  Employee skeu2 = ef.create ("Staf", "Megadewi", "Staf Keuangan", kis);  Employee ssdm1 = ef.create ("Staf", "Trifina", "Staf SDM", kis);  Employee ssdm2 = ef.create ("Staf", "Kanda", "Staf SDM", kis);  Employee ssdm3 = ef.create ("Staf", "Arjuna", "Staf SDM", kis);  direktur.addChild(mankeu); direktur.addChild(mansdm);  mankeu.addChild(kaakun); mankeu.addChild(kaaudt);  kaakun.addChild(skeu1);  kaaudt.addChild(skeu2);  mansdm.addChild(kapsdm); mansdm.addChild(kaprom);  kapsdm.addChild(ssdm1); kapsdm.addChild(ssdm2);  kaprom.addChild(ssdm3);  direktur.traverseEmp();  InputStreamReader values = new InputStreamReader (System.in);  BufferedReader br = new BufferedReader(values);  do {  System.out.print("\nInput kenaikan premi [0..1] : ");  String valueStr = br.readLine();  if (valueStr.equalsIgnoreCase("00"))  break;  kis.setState(Float.parseFloat(valueStr));  direktur.traverseEmp();  } while(true);  }  }   1. Eksekusi kode program. |
| *Apa yang bisa Anda simpulkan dengan menggunakan Singleton Pattern ?*  Memastikan sebuah kelas hanya memiliki sebuah instansi atau kelas.  Buat objek kelas tsb secara static.  1. Menciptakan variable kelas.  2. Buat private kontruktor supaya tidak bisa diakses di kelas lain.  3. Menciptakan method. Memastikan hanya satu objek tercipta. |

**No Kelompok :**

**Anggota :**

|  |  |
| --- | --- |
| **NPM Anggota** | **Nama Anggota** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Lampiran Kode Program pada tutorial 11.4 - Composite Observer Pattern**

|  |
| --- |
| 1. Kelas **Employee**   public abstract class Employee {  public static StringBuffer space = new StringBuffer();  // atribut-atribut kelas Employee sebagai sebuah konsep  protected String role; // jabatan employee  protected String name;  protected float asuransi; // premi asuransi employee  public Employee (String nm, String rl, float asr) {  name = nm; role = rl;  asuransi = asr;  }  public Employee (String nm, String rl, float asr, Asuransi mod) {  name = nm; role = rl;  asuransi = asr;  model = mod;  model.attach(this); // Employee sebagai observers mendaftar  // ke Asuransi (sebagai subject)  }  // fungsi-fungsi Employee sebagai abstract component  // dalam pattern composite. Override di kelas turunan  public abstract void addChild(Employee e);  public abstract Employee getChild(int i);  public abstract void traverseEmp();  // fungsi-fungsi getter kelas Employee sebagai sebuah konsep  public String getRole() { return role; }  public String getName() { return name; }  public float getAsuransi() { return asuransi ; }  /////////////////////////////////////////////////////  // atribut kelas Employee sebagai abstract observer  // dalam pattern observer  private Asuransi model;  /////////////////////////////////////////////////////  /////////////////////////////////////////////////////  // fungsi-fungsi kelas Employee sebagai abstract observer  // dalam pattern observer  public abstract void update(float val);  /////////////////////////////////////////////////////  }   1. Kelas **Manager** dan kelas **Staf**   //Manager.java  import java.util.ArrayList;  import java.util.List;  public class Manager extends Employee {  // atribut kelas Manager sebagai composite class dalam  // pattern composite  List<Employee> employees = new ArrayList<Employee>();  public Manager(String nm, String rl){  super(nm, rl, 1000000);  }  public Manager(String nm, String rl, Asuransi mod){  super(nm, rl, 1000000, mod);  }  // fungsi-fungsi kelas Manager sebagai composite class  // dalam pattern composite  public void addChild(Employee e) { // Catat siapa saja yang  employees.add(e); // menjadi bawahan manager  }  public Employee getChild(int i) { // Cari tahu bawahan untuk  return employees.get(i); // seorang manager  }  public void traverseEmp() { // tampilkan semua employee  System.out.println( Employee.space + getName() + "(" +  getRole() + " - " + getAsuransi() + ")" );  Employee.space.append(" ");  for (Employee e : employees)  {  e.traverseEmp();  }  Employee.space.setLength(Employee.space.length() - 3);  }  ///////////////////////////////////////////////////////  // fungsi kelas Manager sebagai concrete observer class  // pattern observer  public void update(float val) {  asuransi = asuransi + asuransi \* val;  }  ///////////////////////////////////////////////////////  }  //Staf.java  public class Staf extends Employee {  public Staf (String nm, String rl){  super(nm, rl, 750000);  }  public Staf (String nm, String rl, Asuransi mod){  super(nm, rl, 750000, mod);  }  // fungsi-fungsi kelas Staf sebagai leaf class dalam  // pattern composite  public void addChild(Employee employee) {} // do nothing. Kelas Staf  // tidak memiliki bawahan  public Employee getChild(int i) {  return null;  }  public void traverseEmp() {  System.out.println( Employee.space + getName() + "(" +  getRole() + " - " + getAsuransi() + ")" );  }  ///////////////////////////////////////////////////////  // fungsi kelas Staf sebagai concrete observer dalam  // pattern composite  public void update(float val) {  asuransi = asuransi + asuransi \* val;  }  }   1. Kelas **Asuransi** sebagai subject dalam pattern observer   import java.util.Vector;  public class Asuransi {  // atribut kelas Asuransi sebagai subject dalam pattern observer  private Vector<Employee> views;  public Asuransi() {  views = new Vector<Employee>();  }  // fungsi-fungsi kelas Asuransi sebagai subject dalam  // pattern observer  public void attach (Employee obs) {  views.add(obs);  }  public void notifyObserver(float val) {  for (int i=0; i < views.size(); i++)  views.elementAt(i).update(val);  }  public void setState(float val) {  notifyObserver(val);  }  } |
|  |