**Form Diskusi Pertemuan 10**

**SOLID Pattern**

**Tujuan diskusi:**

* Mahasiswa mampu mengimplementasi beberapa pola (pattern) prinsip perancangan dengan menggunakan bahasa Java.

**Tutorial 10.1: SOLID Single Responsibilty Principle**

|  |
| --- |
| 1. Perhatikan kode program yang ada dalam Lampiran 01. Dari kode program terlihat bahwa class **BankAccount** mewadahi dua konsep yang berbeda, yang dijadikan dalam sebuah kelas tunggal. Kedua konsep tersebut adalah rekening tabungan dan rekening deposito.   Prinsip kohesi mengatakan bahwa setiap kelas seyogyanya hanya mengandung satu konsep tunggal saja. Hal ini dipertegaskan kembali dalam prinsip Single Responsibility Principle yang mengatakan “A class should concentrate on doing one thing and one thing only”. Dengan demikian, dalam perancangan kelas yang baik, maka kelas **BankAccount** sebaiknya dipisah menjadi kelas yang menangani tabungan dan kelas yang menangani deposito.  Untuk melakukan pemisahan, kita dapat menggunakan antarmuka (interface atau kelas abstract) yang memungkinkan penambahan jenis kelas lain yang mengimplementasikan antarmuka tersebut.   1. Buatlah project dengan nama **SRP** 2. Buatlah interface **BankAccount**, dengan kode   public interface BankAccount {  public void openSavingAccount(double bal);  public void storeCoins (Coins amt);  public void withdrawal (double amt);  public void displayBalance();    public void openDeposit(double dep);  public void displayDeposit();  }   1. Buatlah kelas **SavingsAccount** dan kelas **DepositAccount,** dengan kode program berikut ini:   //SavingsAccount.java  public class SavingsAccount implements BankAccount {  private String number;  private double balance;    public SavingsAccount (String num) {  number = num;  }    public void openSavingAccount(double bal) {  balance = bal;  displayBalance();  }    public void storeCoins (Coins c) {  balance = balance + c.getValue();  displayBalance();  }    public void withdrawal (double amt) {  balance = balance - amt;  displayBalance();  }    public void displayBalance() {  System.out.println("Rekening Tabungan No " + number +  " saldo tabungan : " + balance);  }    public void openDeposit(double dep) {}  public void displayDeposit() {}  }  //DepositAccount.java  public class DepositAccount implements BankAccount {  private String number;  private double deposit;    public DepositAccount (String num) {  number = num;  }    public void openSavingAccount(double bal) {}  public void storeCoins (Coins c) {}  public void withdrawal (double amt) {}  public void transfer(BankAccount b, double amt) {}  public void displayBalance() {}    public void openDeposit(double dep) {  deposit = dep;  displayDeposit();  }    public void displayDeposit() {  System.out.println("Rekening Deposito No " + number +  " saldo deposito : " + deposit);  }  }   1. Buatlah kelas **SRPTest** dengan kode berikut ini:   public class SRPTest {  public static void main(String[] args) {    BankAccount b = new SavingsAccount("1000");  BankAccount c = new SavingsAccount("1100");  BankAccount d = new DepositAccount("1200");  Coins cc = new Coins(5000);  b.openSavingAccount(10000);  c.openSavingAccount(20000);  d.openDeposit(1000000);  b.storeCoins(cc);  b.withdrawal(2500);  }  }     1. Eksekusi program.      1. Amati kode program di kelas **SavingsAccount** dan kelas **DepositAccount.** Apa yang terlihat dalam kode program di kelas-kelas tersebut ? |
|  |

**Tutorial 10.2: SOLID Interface Segregation Principle**

|  |
| --- |
| 1. Dari tutorial 10.1 terlihat bahwa pada kelas **SavingsAccount** dan kelas **DepositAccount** terdapat fungsi-fungsi kosong. Fungsi kosong menunjukkan bahwa fungsi tersebut tidak relevan pada kelas tersebut. Mengapa bisa terjadi? Hal ini terjadi karena kedua kelas harus mengimplementasikan interface yang masih mengakomodasi dua konsep, yaitu konsep rekening tabungan dan rekening deposito, sehingga terlihat interface **BankAccount** merupakan sebuah interface yang kompleks (fat interface).   Interface Segregation Principle mengatakan “Clients should not be forced to depend upon interfaces that they do not use.” Hal ini mengandung arti bahwa interface harus dibuat minimal sesuai dengan kebutuhan kelas. Dengan demikian, dalam perancangan kelas yang baik, maka interface **BankAccount** sebaiknya dipisah menjadi interface yang khusus menangani tabungan dan kelas yang menangani deposito.   1. Buatlah project dengan nama **ISP** 2. Buatlah interface **ISavingsAccount** dan **IDepositAccount**, dengan kode   //ISavingsAccount.java  public interface ISavingsAccount {  public void openSavingAccount(double bal);  public void storeCoins (Coins amt);  public void withdrawal (double amt);  public void displayBalance();  }  //IDepositAccount.java  public interface IDepositAccount {  public void openDeposit(double dep);  public void displayDeposit();  }   1. Buatlah kelas **SavingsAccount** dan kelas **DepositAccount,** dengan kode program berikut ini:   //SavingsAccount.java  public class SavingsAccount implements ISavingsAccount {  private String number;  private double balance;    public SavingsAccount (String num) {  number = num;  }    public void openSavingAccount(double bal) {  balance = bal;  displayBalance();  }    public void storeCoins (Coins c) {  balance = balance + c.getValue();  displayBalance();  }    public void withdrawal (double amt) {  balance = balance - amt;  displayBalance();  }    public void displayBalance() {  System.out.println("Rekening Tabungan No " + number +  " saldo tabungan : " + balance);  }    }  //DepositAccount.java  public class DepositAccount implements IDepositAccount {  private String number;  private double deposit;    public DepositAccount (String num) {  number = num;  }    public void openDeposit(double dep) {  deposit = dep;  displayDeposit();  }    public void displayDeposit() {  System.out.println("Rekening Deposito No " + number +  " saldo deposito : " + deposit);  }  }   1. Buatlah kelas **ISPTest** dengan kode berikut ini:   public class ISPTest {  public static void main(String[] args) {    ISavingsAccount b = new SavingsAccount("1000");  ISavingsAccount c = new SavingsAccount("1100");  IDepositAccount d = new DepositAccount("1200");  Coins cc = new Coins(5000);  b.openSavingAccount(10000);  c.openSavingAccount(20000);  d.openDeposit(1000000);  b.storeCoins(cc);  b.withdrawal(2500);  }  }     1. Eksekusi program.      1. Amati kode program di kelas **SavingsAccount** dan kelas **DepositAccount.** Apa perbedaan kelas di tutorial 10.1 dan 10.2 ? |
|  |

**Tutorial 10.3: Non SOLID Patterns**

|  |
| --- |
| 1. Buatlah project dengan nama **NonOLD** 2. Tutorial ini akan mencoba menunjukkan bagaimana pengkodean tanpa menggunakan pattern SOLID, secara khusus prinsip **DIP, OCP dan LSP** . Kode program dibuat berdasarkan ide struktur berikut ini:     Gambar 1a. High Level Module bergantung pada Low Level Module  Gambar 1b. Kelas SavingsAccount bergantung pada Coins, Notes dan Cheque   1. Dengan menggunakan ide tutorial sebelumnya, saat ini diinginkan bahwa uang yang disimpan bukan hanya coins saja melainkan juga uang kertas (Notes). 2. Buatlah kelas **Notes** berikut ini.   //Notes.java  public class Notes {  private double value;  public Notes (double aValue) {  value = aValue;  }  public double getValue() {  return value;  }  }   1. Untuk mengakomodasi penambahan jenis uang ini, maka modifikasi interface **ISavingsAccount** dan kelas **SavingsAccount**. Tambahan kode adalah **berwarna merah.**   //ISavingsAccount.java  public interface ISavingsAccount {  public void openSavingAccount(double bal);  public void storeCoins (Coins amt);  **public void storeNotes (Notes amt);**  public void withdrawal (double amt);  public void displayBalance();    }  //SavingsAccount.java  public class SavingsAccount implements ISavingsAccount {  private String number;  private double balance;    public SavingsAccount (String num) {  number = num;  }    public void openSavingAccount(double bal) {  balance = bal;  displayBalance();  }    public void storeCoins (Coins c) {  balance = balance + c.getValue();  displayBalance();  }  **public void storeNotes (Notes c) {**  **balance = balance + c.getValue();**  **displayBalance();**  **}**  public void withdrawal (double amt) {  balance = balance - amt;  displayBalance();  }    public void displayBalance() {  System.out.println("Rekening Tabungan No " + number +  " saldo tabungan : " + balance);  }  }   1. Buatlah kelas **NonOLDTest** dengan kode berikut ini:   public class NonOLDTest {  public static void main(String[] args) {  // TODO code application logic here  ISavingsAccount b = new SavingsAccount("1000");  ISavingsAccount c = new SavingsAccount("1100");  IDepositAccount d = new DepositAccount("1200");  Coins cc = new Coins(500);  **Notes nn = new Notes(1000);**  b.openSavingAccount(10000);  c.openSavingAccount(20000);  d.openDeposit(1000000);  b.storeCoins(cc);  b.withdrawal(2500);  **c.storeNotes(nn);**  }  }   1. Eksekusi program.      1. Buatlah sebuah kelas baru, yaitu kelas **Cheque,**yang memiliki atribut **value** bertipe **double**. Implementasi kelas ini dengan menggunakan pola kelas **Notes** yang ada pada langkah 4. 2. Supaya uang Cheque bisa disimpan menggunakan kelas **SavingsAccount**, lakukan modifikasi pada interface **ISavingsAccount** dan kelas **SavingsAccount** dengan menambah fungsi **storeCheque(Cheque c)**. 3. Lakukan pengujian dengan kode program berikut ini:   public class NonOLDTest {  public static void main(String[] args) {  ISavingsAccount b = new SavingsAccount("1000");  ISavingsAccount c = new SavingsAccount("1100");  Coins cc = new Coins(500);  Notes nn = new Notes(1000);  **Cheque ch = new Cheque(10000);**  b.openSavingAccount(10000);  c.openSavingAccount(20000);  b.storeCoins(cc);  c.storeNotes(nn);  **b.storeCheque(ch);**  }    }   1. Eksekusi kode program, dan jawab pertanyaan di bawah ini. |
| 1. Kelas **SavingsAccount** yang merupakan high level module bergantung pada kelas-kelas lain (**Coins**, **Notes**, dan **Cheque**). Apabila ada penambahan lain maka …… [tuliskan hasil pengamatan Anda] |

**Tutorial 10.4: Dependency Inversion Principle, Open Closed Principle & Liskov Substitution Principle**

|  |
| --- |
| 1. Beberapa pattern akan digunakan untuk memperbaiki kode program yang dibuat pada tutorial 10.3 di atas, yaitu **Dependency Inversion Principle (DIP)**, **Open Closed Principle (OCP)** dan **Liskov Substitution Principle (LSP)**. 2. Dependency Inversion Principle menyatakan menyatakan :   High level modules should not depend upon low level modules. Both should depend upon abstractions.   1. Tutorial 10.4 ini akan mewujudkan prinsip **DIP**, yang memodifikasi gambar 1a dan 1b di atas menjadi struktur gambar 2a dan 2b berikut ini. Untuk mengurangi kebergantungan dari high level module (kelas **SavingsDeposit** ) terhadap kelas lain (**Coins**, **Notes**, dan **Cheque**), maka dibuat antarmuka yang merupakan abstraksi dari kelas-kelas lain tersebut. Abstraksi dapat dibuat dengan menggunakan interface atau kelas induk. Dalam tutorial ini akan digunakan interface **Currency.**       Gambar 2a. High Level Module bergantung pada abstraksi dari Low Level Modul  Gambar 2b. Kelas SavingsAccount bergantung pada abstraksi (interface) dari Coins, Notes dan Cheque   1. Selain itu, tutorial 10.4 ini juga mengimplementasikan **OCP**. Prinsip ini menyatakan : 2. Sebuah kelas terbuka untuk perluasan (*open for extension*) tetapi tertutup untuk modifikasi (*closed for modification*). 3. Sebuah fungsionalitas baru ditambahkan dengan membuat kelas, method atau field baru, bukan menggantikan kode yang sudah teruji 4. Dari tutorial 10.3 dapat dilihat, apabila ada penambahan kelas bentuk jenis uang lain, maka **SavingsDeposit** harus selalu diubah (dimodifikasi) atau kode programnya harus diganti (berarti *open for modification*). Hal ini menunjukkan bahwa tutorial 10.3 melanggar prinsip OCP (lihat poin 4a di atas). Perubahan yang diperbolehkan adalah melalui perluasan (extension) yang dapat diwujudkan dengan pewarisan, dan dikombinasikan dengan prinsip (Liskov Substitution Prinsiple (LSP).   Prinsip **LSP** menyatakan : “sebuah fungsi yang menggunakan referensi sebuah kelas induk harus dapat digunakan juga oleh objek yang berasal dari kelas turunan dari kelas induk tersebut”   1. Buatlah project dengan nama **SolidOLD** 2. Buatlah interface **ICurrency**, dengan kode   public interface ICurrency {  public double getValue();  }   1. Buatlah kelas **Coins** dan **Notes,** dengan kode program berikut ini:   //Coins.java  public class Coins implements ICurrency {  private double value;  public Coins(double aValue) {  value = aValue;  }  public double getValue() {  return value;  }  }  //Notes.java  public class Notes implements ICurrency {  private double value;  public Notes(double aValue) {  value = aValue;  }  public double getValue() {  return value;  }  }   1. Buatlah interface **ISavingsAccount** dan kelas **SavingsAccount**, dengan kode berikut ini:   //ISavingsAccount.java  public interface ISavingsAccount {  public void openSavingAccount(double bal);  **public void store (ICurrency amt);**  public void withdrawal (double amt);  public void displayBalance();    }  //SavingsAccount.java  public class SavingsAccount implements ISavingsAccount {  private String number;  private double balance;    public SavingsAccount (String num) {  number = num;  }    public void openSavingAccount(double bal) {  balance = bal;  displayBalance();  }    **public void store (ICurrency c) {**  **balance = balance + c.getValue();**  **displayBalance();**  **}**    public void withdrawal (double amt) {  balance = balance - amt;  displayBalance();  }    public void displayBalance() {  System.out.println("Rekening Tabungan No " + number +  " saldo tabungan : " + balance);  }    }  Penggunaan Parameter bertipe **ICurrency** menunjukkan perwujudan LSP. Dengan menggunakan parameter ini, maka **amt** dapat diisi atau digantikan dengan setiap objek dari kelas yang menimplementasikan **ICurrency.**   1. Buatlah kelas **SolidOLDTest** dengan kode berikut ini:   public class SOLID01Test {  public static void main(String[] args) {  ISavingsAccount b = new SavingsAccount("1000");  ISavingsAccount c = new SavingsAccount("1100");  IDepositAccount d = new DepositAccount("1200");  **Coins cc = new Coins(500);**  **Notes nn = new Notes(1000);**  b.openSavingAccount(10000);  c.openSavingAccount(20000);  d.openDeposit(1000000);  **b.store(cc);**  b.withdrawal(2500);  **c.store(nn);**  }  }   1. Eksekusi program.      1. Modifikasi kelas **Cheque,** yang mengimplementasikan **ICurrency.** Kelas ini memiliki atribut **value** bertipe **double**. Implementasi kelas ini dengan menggunakan pola kelas **Coins** yang ada pada langkah 8. 2. Supaya **Cheque** bisa disimpan **SavingsAccount**, apakah kita perlu melakukan modifikasi terhadap fungsi **store** di interface **ISavingsAccount** dan kelas **SavingsAccount** ? Jawaban **tidak** menunjukkan bahwa prinsip OCP telah berhasil diimplementasikan dengan baik. Artinya fungsi **store** tidak mengalami perubahan (closed for modification) untuk setiap penambahan kelas yang mengimplementasikan **ICurrency.** Sementara itu, kelas implementor **ICurrency** dapat bertambah terus (open for extension) yang mengakibatkan fungsionalitas dari fungsi **store** bertambah, tanpa harus melakukan perubahan terhadap kode program. 3. Lakukan pengujian dengan kode program berikut ini:   public class SolidOLDTest {  public static void main(String[] args) {  ISavingsAccount b = new SavingsAccount("1000");  ISavingsAccount c = new SavingsAccount("1100");  IDepositAccount d = new DepositAccount("1200");  Coins cc = new Coins(500);  Notes nn = new Notes(1000);  **Cheque ch = new Cheque(10000);**  b.openSavingAccount(10000);  c.openSavingAccount(20000);  d.openDeposit(1000000);  b.store(cc);  b.withdrawal(2500);  c.store(nn);  **c.store(ch);**  }  }   1. Eksekusi kode program. |
| *Tuliskan apa harus Anda lakukan, jika ada penambahan jenis uang lain*  *Tuliskan kesimpulan Anda …* |

**No Kelompok :**

**Anggota :**

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

**LAMPIRAN 1**

//Coins.java

public class Coins {

private double value;

public Coins(double aValue) {

value = aValue;

}

public double getValue() {

return value;

}

}

//BankAccount.java

public class BankAccount {

private String number;

private double balance;

private double deposit;

public BankAccount (String num) {

number = num;

}

public void openSavingAccount(double bal) {

balance = bal;

displayBalance();

}

public void storeCoins (Coins c) {

balance = balance + c.getValue();

displayBalance();

}

public void withdrawal (double amt) {

balance = balance - amt;

displayBalance();

}

public void displayBalance() {

System.out.println("Rekening Tabungan No " + number +

" saldo tabungan : " + balance);

}

public void openDeposit(double dep) {

deposit = dep;

displayDeposit();

}

public void displayDeposit() {

System.out.println("Rekening Deposito No " + number +

" saldo deposito : " + deposit);

}

}