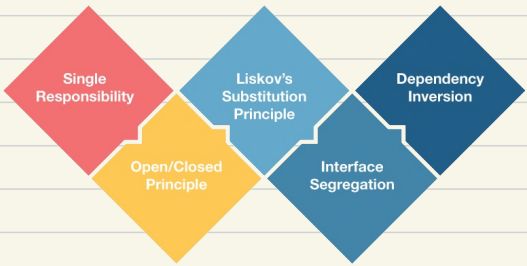
Tue 13th Dec

→ Definition :

S.O.L.I.D.



" Solid components should be open for EXTENSION, but closed for MODIFICATION ".

Rule 1 : Single Responsibility Principle (SRP)

e.g.: train lines / xxw

- "Every software component should have one and only one ~~responsibility~~ reason to change."

component → class, method, or module

- Cohesion : "Cohesion is the degree to which the various parts of a software component are related".

```
public class Square {

    int side = 5;

    public int calculateArea() {
        return side * side;
    }

    public int calculatePerimeter() {
        return side * 4;
    }

    public void draw() {
        if (highResolutionMonitor) {
            // Render a high resolution image of a square
        } else {
            // Render a normal image of a square
        }
    }

    public void rotate(int degree) {
        // Rotate the image of the square clockwise to
        // the required degree and re-render
    }
}
```

High level of cohesion

High level of cohesion

When there're this discrepancies in cohesion inside a class we MUST split them in different ones !

```java
public class Square {

      int side = 5;

      public int calculateArea() {
            return side * side;
      }

      public int calculatePerimeter() {
            return side * 4;
      }
}
```

```java
public class SquareUI {

      public void draw() {
            if (highResolutionMonitor) {
                  // Render a high resolution image of a square
            } else {
                  // Render a normal image of a square
            }
      }

      public void rotate(int degree) {
            // Rotate the image of the square clockwise to
            // the required degree and re-render
      }

}
```

Now, "Higher Cohesion helps attain better adherance to the SRP."

- Coupling: "Coupling is defined as level of inter dependency between various software components"

```java
public class Student {

      private String studentId;
      private Date studentDOB;
      private String address;

      public void save() {
            // Serialize object into a string representation
            String objectStr = MyUtils.serializeIntoAString(this);
            Connection connection = null;
            Statement stmt = null;
            try {
            Class.forName("com.mysql.jdbc.Driver");
            connection =
            DriverManager.getConnection("jdbc:mysql://localhost:3306/
            MyDB", "root", "password");
            stmt = connection.createStatement();
            stmt.execute("INSERT INTO STUDENT VALUES (" + objectStr +
            ")");
            } catch (Exception e) {
                        e.printStackTrace();
            }
      }

      public String getStudentId() {
            return studentId;
      }

      public void setStudentId(String studentId) {
            this.studentId = studentId;
      }

      ...
      ...
      ....
      ......
}
```

TIGHT COUPLING IS BAD IN SOLID

The Student class should not be made cognizant of the low level details related to dealing with the back end database.

The Student class should deal with IDs, names, surnames, etc ...

Solution :

```
public class Student {

      private String studentId;
      private Date studentDOB;
      private String address;

      public void save() {
            new StudentRepository().save(this);
      }

      public String getStudentId() {
            return studentId;
      }

      public void setStudentId(String studentId) {
            this.studentId = studentId;
      }

      ...
      ...
      ....
      ......
}
```

Responsibility: Handle core student profile data

```
public class StudentRepository {
      public void save(Student student) {
            // Serialize object into a string representation
            String objectStr = MyUtils.serializeIntoAString(student);
            Connection connection = null;
            Statement stmt = null;
            try {
                  Class.forName("com.mysql.jdbc.Driver");
                  connection =
                  DriverManager.getConnection("jdbc:mysql://localhost:
                  3306/MyDB", "root", "password");
                  stmt = connection.createStatement();
                  stmt.execute("INSERT INTO STUDENT VALUES (" +
                  objectStr + ")");
            } catch (Exception e) {
                  e.printStackTrace();
            }
      }
}
```

Responsibility: Handle Database Operations

Fri 16th Dec

Rule 2 : Open / Close Principle → e.g.: Nintendo Wii ↘ closed for modification in hardware; open for extensions (add car wheel into controller, gun accessory, ...)

- "Software components should be closed for modification, but open for extension".

- New features getting added to the software component, should NOT have to modify existing code.

- A software component should be extendable to add a new feature or to add a new behavior to it.

```
public class InsurancePremiumDiscountCalculator {
      public int
      calculatePremiumDiscountPercent(HealthInsuranceCustomerProfile customer)
      {
            if (customer.isLoyalCustomer()) {
                  return 20;
            }
            return 0;
      }
}
```

↘ function that handles the promotion

↘ passes as parameter only the Health type

How to ADDED ?

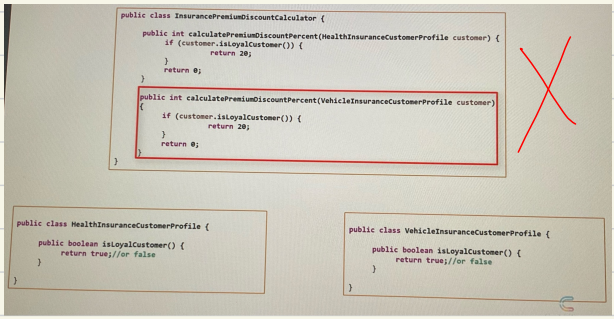
↘ currently only type

```
public class HealthInsuranceCustomerProfile {

      public boolean isLoyalCustomer() {
            return true;//or false
      }
}
```

↘ new type of customer profile

```
public class VehicleInsuranceCustomerProfile {

      public boolean isLoyalCustomer() {
            return true;//or false
      }
}
```

NOT SOLID solution:

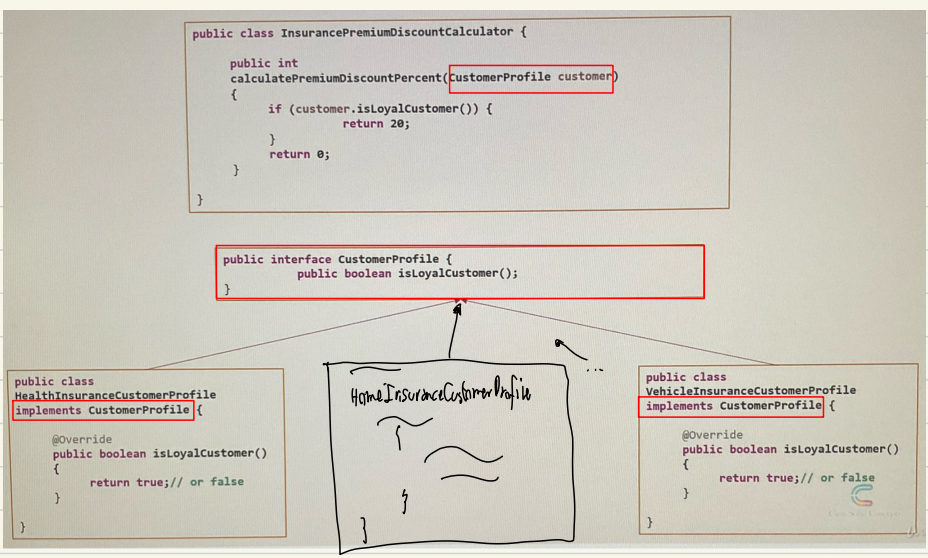


```
public class InsurancePremiumDiscountCalculator {

    public int calculatePremiumDiscountPercent(HealthInsuranceCustomerProfile customer) {
        if (customer.isLoyalCustomer()) {
            return 20;
        }
        return 0;
    }

    public int calculatePremiumDiscountPercent(VehicleInsuranceCustomerProfile customer)
    {
        if (customer.isLoyalCustomer()) {
            return 20;
        }
        return 0;
    }
}
```

```
public class HealthInsuranceCustomerProfile {

    public boolean isLoyalCustomer() {
        return true;//or false
    }
}
```

```
public class VehicleInsuranceCustomerProfile {

    public boolean isLoyalCustomer() {
        return true;//or false
    }
}
```

Best optimal solution with SOLID PRINCIPLE:



```
public class InsurancePremiumDiscountCalculator {

    public int
    calculatePremiumDiscountPercent(CustomerProfile customer)
    {
        if (customer.isLoyalCustomer()) {
            return 20;
        }
        return 0;
    }
}
```

```
public interface CustomerProfile {
        public boolean isLoyalCustomer();
}
```

```
public class
HealthInsuranceCustomerProfile
implements CustomerProfile {

    @Override
    public boolean isLoyalCustomer()
    {
        return true;// or false
    }
}
```

HomeInsuranceCustomerProfile

```
public class
VehicleInsuranceCustomerProfile
implements CustomerProfile {

    @Override
    public boolean isLoyalCustomer()
    {
        return true;// or false
    }
}
```

- We've created a new Interface and for each class (Health and Vehicle) it implements this CustomerProfile.
- In the InsurancePremiumDiscountCalculator, by passing the interface customerProfile it'll accept any type, either Health, Vehicle or new ones, Home, for instance.

Concept of inheritance

Gasoline —IS-A→ FUEL

Tesla —IS-A→ CAR

Mon 26th Dec

Rule 3 : Liskov Substitution Principle → e.g.: "The Is-A way of thinking"

- "Objects should be replaceable with their subtypes without affecting the correctness of the program"

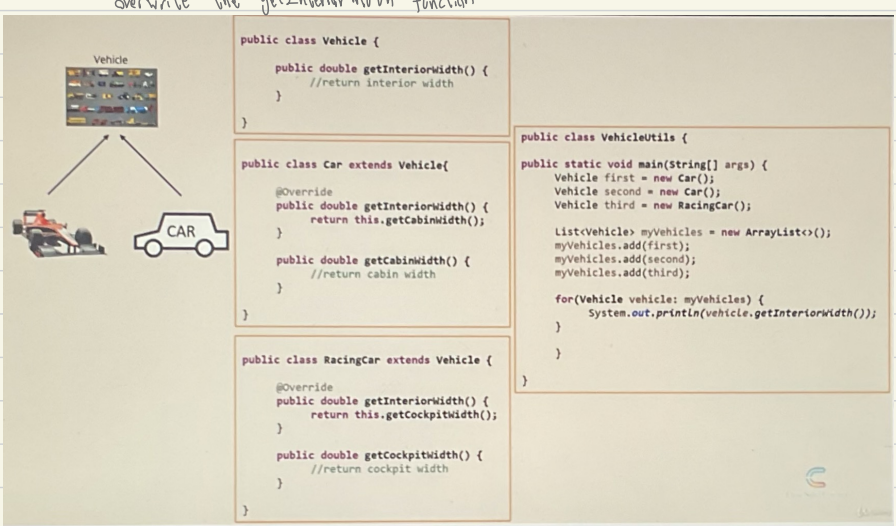
When there's differencies in hierarchy classes we can:

SOLUTION 1:

BEFORE: CAR ↑ RACING CAR

AFTER: VEHICLE ↑ RACING CAR, CAR

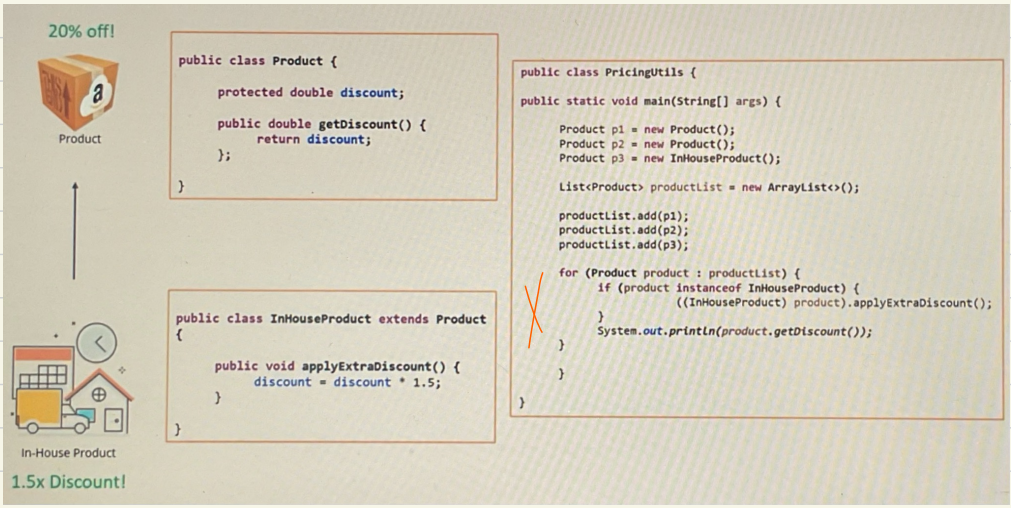
Eg 1: For 2 cars, a general car and a F1 RacingCar we can't measure the width of the interior because the second one don't acually have an interior, but a cabin. Then knowing that both are cars, we've need to create an upperlevel class called Vehicle and for each type overwrite the getInteriorWidth function

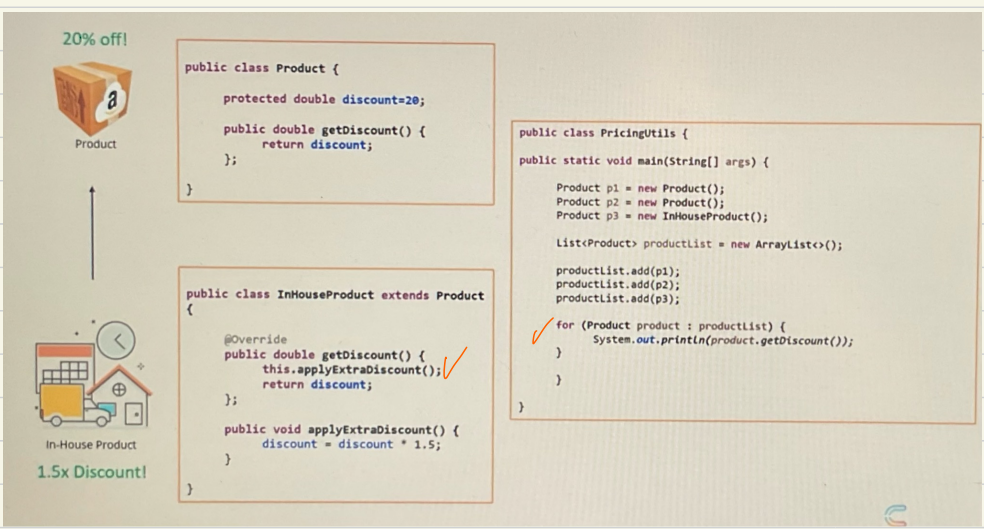


Break the hierarchy if it fails the substitution test.

SOLUTION 2: "Tell, don't ask".

NOT A SOLID PRINCIPLE:





Wed 28th Dec

Rule 4 : Interface Segregation Principle → e.g.:

```java
public interface IPrint {
    public void print();
    public void getPrintSpoolDetails();
}
```

```java
public interface IScan {
    public void scan();
    public void scanPhoto();
}
```

```java
public interface IFax {
    public void fax();
    public void internetFax();
}
```

```java
public class XeroxWorkCentre implements IPrint,IScan,IFax {

    @Override
    public void print() {
    // Assume real printing code
    }

    @Override
    public void getPrintSpoolDetails() {
    // Assume real code that gets spool details
    }

    @Override
    public void scan() {
    // Assume real code for scanning
    }

    @Override
    public void scanPhoto() {
    // Assume real code for photos scans
    }

    @Override
    public void fax() {
    // Assume real code for fax
    }

    @Override
    public void internetFax() {
    // Assume real code for internet fax
    }

}
```

```java
public class HPPrinterNScanner implements IPrint,IScan {

    @Override
    public void print() {
    // Assume real printing code
    }

    @Override
    public void getPrintSpoolDetails() {
    // Assume real code that gets spool details
    }

    @Override
    public void scan() {
    // Assume real code for scanning
    }

    @Override
    public void scanPhoto() {
    // Assume real code for photos scans
    }

}
```

```java
public class CanonPrinter implements IPrint {

    @Override
    public void print() {
    // Assume real printing code
    }

    @Override
    public void getPrintSpoolDetails() {
    // Assume real code that gets spool details
    }

}
```

- Techniques to identify violations of ISP:

1. Fat Interfaces
2. Interfaces with Low Cohesion
3. Empty Method Implementations

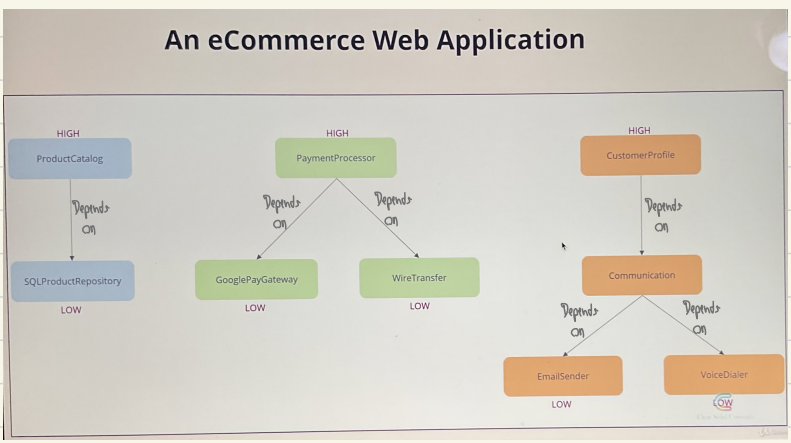
- "SOLID Principles complement each other, and work together in unison, to achieve the common purpose of a well-design software."

Wed 1st Feb

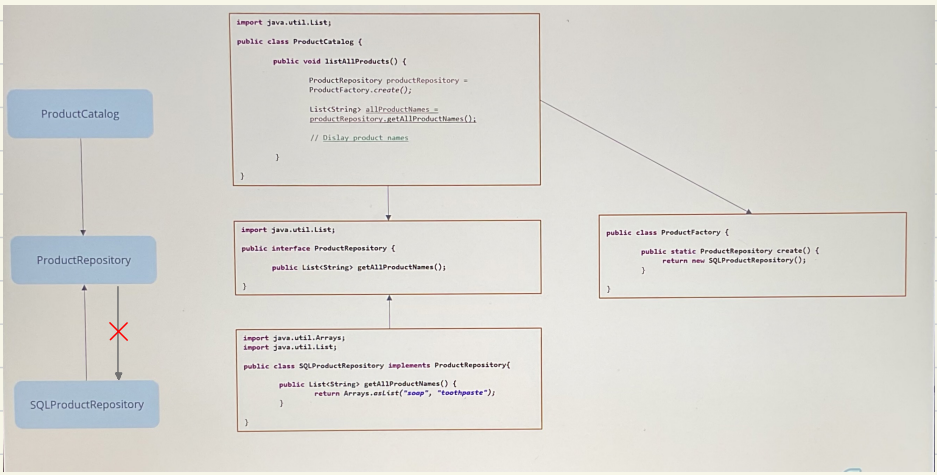
Rule 5 : Dependency Inversion Principle → e.g.:

- "High-level modules should not depend on low-level modules. Both should depend on abstractions".
- "Abstractions should not depend on details. Details should depend on abstractions".

Depend> on



e.g.:



Dependency Injection

Eg.: