Design Patterns

Engr. Abdul-Rahman Mahmood

DPM, MCP, QMR(ISO9001:2000)

armahmood786@yahoo.com
alphapeeler.sf.net/pubkeys/pkey.htm
pk.linkedin.com/in/armahmood
www.twitter.com/alphapeeler
www.facebook.com/alphapeeler
abdulmahmood-sss
alphasecure
armahmood786@hotmail.com
http://alphapeeler.sf.net/me
alphasecure@gmail.com
http://alphapeeler.sourceforge.net
http://alphapeeler.tumblr.com
armahmood786@jabber.org
alphapeeler@aim.com
mahmood_cubix
48660186
alphapeeler@icloud.com
http://alphapeeler.sf.net/acms/

SOLID Principles

- Single responsibility principle
  - A class should have only a single responsibility.
- Open/closed principle
  - Software modules should be open for extension, but closed for modification.
- Liskov substitution principle
  - Objects should be replaceable with instances of their subtypes without altering correctness of that program.
- Interface segregation principle
  - Many client-specific interfaces are better than one general-purpose (monolithic) interface.
- Dependency inversion principle
  - Write code that depends upon abstractions rather than concrete details.

The Open-Closed Principle (OCP)

- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modi cation.
- OR
- To change behavior, add new code rather than changing existing code.
- **How can we confirm to OCP principle?**
- Allow the modules (classes) to depend on the **abstractions**, there by new features can be added by creating new extensions of these abstractions.

e.g. Client Server

- With regards to the Client, the following design does not conform to the OCP.

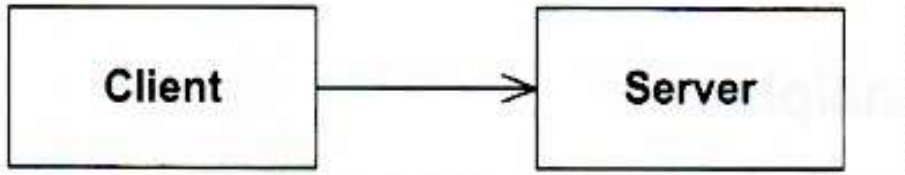
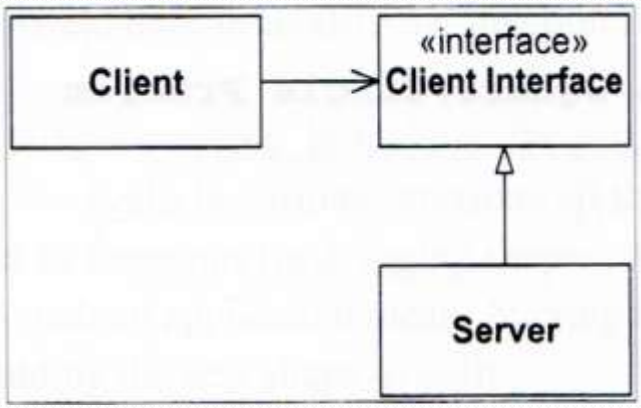


Figure 9-1 Client is not open and closed

- If we want the Client to use a different Server, we must change the Client. However, the following design resolves this problem:



Ex. 1: Calculating Area

- Let's say that we've got a Rectangle class:

```
public class Rectangle {
    public double Width { get; set; }
    public double Height { get; set; }
}
```

- Requirement 1: build an application that can calculate the total area of a ***collection of rectangles***.

```
public class AreaCalculator {
    public double Area(Rectangle[] shapes) {
        double area = 0;
        foreach (var shape in shapes) {
            area += shape.Width*shape.Height;
        }
        return area;
    }
}
```

Calculating Area

- Requirements #2: extend it so that it could calculate the area of not only rectangles but of circles as well.
- Solution: change AreaCalculator() to accept a collection of objects instead of Rectangle type only.

```
public double Area(object[] shapes) {
    double area = 0;
    foreach (var shape in shapes) {
        if (shape is Rectangle) {
            Rectangle rectangle = (Rectangle) shape;
            area += rectangle.Width*rectangle.Height;
        } else {
            Circle circle = (Circle)shape;
            area += circle.Radius * circle.Radius * Math.PI;
        }
    }
    return area;
}
```

Calculating Area

- Requirements #3: Application should also calculate area of triangles and it shouldn't be very hard, is it?”
- Problem : AreaCalculator() isn’t **closed for modification** as we need to change it in order to extend it. Or it isn’t **open for extension**.
- **A solution that abides by the Open/Closed Principle :** create a base class for both rectangles and circles as well as any other shapes that Aldford can think of which defines an abstract method for calculating it’s area.

Calculating Area : OCP

```
public abstract class Shape {
    public abstract double Area();
}

public class Rectangle extends Shape {
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area() {
        return Width*Height;
    }
}

public class Circle extends Shape {
    public double Radius { get; set; }
    public override double Area() {
        return Radius*Radius*Math.PI;
    }
}
```

```
public double Area(Shape[] shapes) {
    double area = 0;
    foreach (var shape in shapes) {
        area += shape.Area();
    }
    return area;
}
```

Ex. 2 Personal Loan App

- Requirement: validate & approve personal loans.

```
public class LoanApprovalHandler {
  public void approveLoan(PersonalValidator validator) {
    if ( validator.isValid()) {
      //Process the loan.
    }
  }
}
public class PersonalLoanValidator {
  public boolean isValid() {
    //Validation logic
  }
}
```

Ex. 2 Personal Loan App

- Enhanced Requirement: approve vehicle loans.

```
public class LoanApprovalHandler {
  public void approvePersonalLoan (PersonalLoanValidator validator) {
    if ( validator.isValid()) {
      //Process the loan.
    }
  }
  public void approveVehicleLoan (VehicleLoanValidator validator ) {
    if ( validator.isValid()) {
      //Process the loan.
    }
  }
  // Method for approving other loans.
}
public class PersonalLoanValidator {
  public boolean isValid() {
    //Validation logic
  }
}
public class VehicleLoanValidator {
  public boolean isValid() {
    //Validation logic
  }
}
```

we ended up changing the name of the existing method (approveLoad->approvePersonalLoad) and also adding new methods (approveVehicleLoan) for different types of loan approval. This clearly violates the OCP.

Ex. 2 Personal Loan App

- OCP solution: create Abstract Validator class and Extended to add validators for different loan types

```
public abstract class Validator {
  public boolean isValid();
}
public class PersonalLoanValidator extends Validator {
  public boolean isValid() {
    //Validation logic.
  }
}
public class VehicleLoanValidator extends Validator {
  public boolean isValid() {
    //Validation logic.
  }
}
public class LoanApprovalHandler {
  public void approveLoan(Validator validator) {
    if ( validator.isValid()) {
      //Process the loan.
    }
  }
}
```