Chapter Four

Design Principles

Chapter Outlines

- Introduction to Design Principles
- Common Design Principles
  - Keep It Simple, Stupid (KISS)
  - Don’t Repeat Yourself Principle (DRY)
  - Complete and Consistent (C&C)
- Famous Design Principles (S.O.L.I.D.)
  - Single Responsibility Principle (SRP)
  - Open-Closed Principle (OCP)
  - Liskov Substitution Principle (LSP)
  - Interface Segregation Principle (ISP)
  - Dependency Inversion Principle (DIP)

Introduction

- How to decompose a system into objects is one of the hardest parts of the OO design
- It involves many factors that come into play
  - encapsulation, granularity, dependency, flexibility, performance, evolution, reusability, etc.
- They influence the decomposition in the conflicting ways
- Design principles provide the ways of thinking in OO to define objects

Design Principles

- Design principle is a basic tool or technique that can be applied to designing or writing code to make that code more maintainable, flexible, and extensible
- Modern information systems will adopt more than one design principle
- It is the code-of-practice (CoP) for developing a large scale system

Common Design Principles

- There are many principles can be applied when developing a system
- Common design principles are:
  - KISS, DRY, C&C, etc.
- Famous design principles are:
  - SOLID: SRP, OCP, LSP, ISP, DIP

Keep It Simple, Stupid Principle

- KISS is an acronym for ***Keep it Simple, Stupid***
  - “Keep it Simple, Silly”, “Keep it Short, Simple”, Keep it Small, Simple”, “Keep it Simple, Straightforward”, etc.
- Most of the systems work best if they are kept simple rather than made complicated
- Simplicity should be a key goal in design and unnecessary complexity should be avoided
- Short program is always easier to debug than a lengthy program

Don’t Repeat Yourself Principle

- DRY: abstracting out the duplicated codes
- It looks pretty straight-forward but turns out to be critical in coding for easy to maintain and ***reuse***
- To have each piece of information and behavior in a single sensible place of the system
- To centralize the duplicate codes in a unique place for integrity and apply changes for all

DRY Example

- Duplicated codes appeared in the following transactions (TXN1: monthly interest, TXN2: monthly service charge)

```
public class TxnOne {
  public TxnOne(double balance, double interest) {
    balance = balance * (1 + interest);
    DecimalFormat df = new DecimalFormat("#,###,##0.00");
    System.out.println("Balance: " + df.format(balance));
  }
}
```

```
public class TxnTwo {
  public TxnTwo(double balance, double serviceCharge) {
    balance = balance - serviceCharge;
    DecimalFormat df = new DecimalFormat("#,###,##0.00");
    System.out.println("Balance: " + df.format(balance));
  }
}
```

Isolate the Codes

- Create a class to hold the duplicated codes

```
public class Tools {
  public static void printAmount(double balance) {
    DecimalFormat df = new DecimalFormat("#,###,##0.00");
    System.out.println("Balance: " + df.format(balance));
  }
}
```

- We can now focus more on the logic of the transactions

```
public class TxnOne {
  public TxnOne(double b, double i) {
    Tools.printAmount(b * (1 + i));
  }
}
```

```
public class TxnTwo {
  public TxnTwo(double b, double c) {
    Tools.printAmount(b - c);
  }
}
```

Complete and Consistent Principle

- Each class should have exactly one role but similar type of behaviors can also add to that class to make it more complete
- There are always related behaviors that they will exist together in nature
  - Enable/Disable, Getter/Setter, Open/Close, Connect/Disconnect, etc.
  - Key Pressed / Key Released / Key Typed, etc.
  - Mouse Drag / Mouse Drop, Mouse Enter / Mouse Exit, etc.
  - As in the example, we can provide the four behaviors (CRUD) for the *DBProcess* class to make it becomes ***complete***

Complete *DBProcess*

```
public class DBProcess {
 public static void createUser(UserBean bean) {
   DBObject dbObj = new DBObject();
    dbObj.connectDatabase();
    dbObj.create(bean.getId(), bean.getName(), bean.getPassword());
    dbObj.disconnectDatabase();
  }

  public static UserBean readUser(UserBean bean) {
    DBObject dbObj = new DBObject();
    dbObj.connectDatabase();
    UserBean userBean = dbObj.read(bean.getId());
    dbObj.disconnectDatabase();
    return userBean;
  }

```

Complete DBProcess (cont.)

```
    public static void updateUser(UserBean bean) {
      DBObject dbObj = new DBObject();
      dbObj.connectDatabase();
      dbObj.update(bean.getId(), bean.getName(), bean.getPassword());
      dbObj.disconnectDatabase();
    }

    public static void deleteUser(UserBean bean) {
      DBObject dbObj = new DBObject();
      dbObj.connectDatabase();
      dbObj.delete(bean.getId());
      dbObj.disconnectDatabase();
    }
  }
```

Coding Consistently

- The naming convention of the methods should be in the same manner
- The parameters they take should be in the same order
- The naming for methods and variables should be meaningful
- Coding inconsistently will create potential risks. It is confusing for people to debug and expand the systems

Inconsistent

- Inconsistent method names

```
public static void createUser(UserBean bean) { … }
public static void query(UserBean bean) { … }
public static void delete_user(UserBean bean) { … }
public static void userUpdate(UserBean bean) { … }
```

- Inconsistent parameter orders

```
public static double plus(double a, double b) { return a + b; }
public static double minus(double b, double a) { return a - b; }
public static double multiply(double a, double b) {return b * a; }
public static double divide(double b, double a) { return b / a; }
```

- Any kind of naming convention or order is fine but they must be consistent and easy to understand
- Write codes for human being (*responsible programming*)

S.O.L.I.D. Design Principles

- S.O.L.I.D is an acronym for the first five object oriented design principles by Robert C. Martin
  - SRP, OCP, LSP, ISP, DIP
- It helps developers to make software that are easy to maintain and extend over time when applying them together
- They also form the best practices to be followed for designing the application classes
  - low coupling, high cohesion, high encapsulation, etc

Introduction to S.O.L.I.D.

- SRP: Single Responsibility Principle
  - A class should have one and only one reason to change, meaning that a class should have only one job
- OCP: Open-Closed Principle
  - Objects or entities should be open for extension but closed for modification
- LSP: Liskov Substitution Principle
  - Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program

Introduction to S.O.L.I.D. (cont.)

- ISP: Interface Segregation Principle
  - A client should never be forced to implement an interface that it doesn’t use or clients shouldn’t be forced to depend on methods they do not use
- DIP: Dependency Inversion Principle
  - Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions

Single Responsibility Principle

- SRP is about breaking a big program into objects, and each object has its own responsibility in the system
    - Breaking a big problem into smaller problems
- Each object is designed to handle just one task
- DRY suggests to isolate the codes when they are duplicated but SRP will isolate the codes because of their responsibilities
- Dismember the codes by types
    - data model, business logic, presentation, control flow, etc.

SRP Steps Example

- Procedural programs always contain many steps within a program, and each step is responded for a task or operation

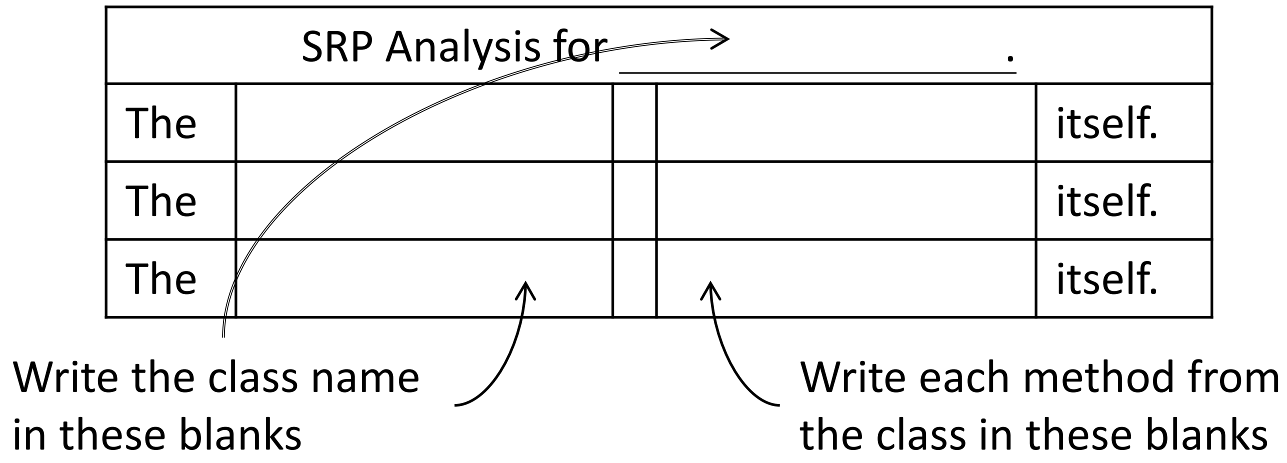
```
public class Procedure1 {
  public Procedure1() {
    step1();
    step2();
  }
  private void step1() { … }
  private void step2() { … }
}
```

```
public class Procedure2 {
  public Procedure2() {
    step2();
    step4();
    step7();
  }
  private void step2() { … }
  private void step4() { … }
  private void step7() { … }
}
```

- We can isolate the steps into different objects
- Some are reusable but more importantly they can be managed and maintained in a better way

Grouping Method

- We often have a single program to handle many tasks. We can't just keep them all in a single object
- Group them as different objects according to their behaviors, role, and responsibilities
- One of the approaches is to fill the following table



SRP Analysis for ____________________.				
The				itself.
The				itself.
The				itself.

Write the class name in these blanks

Write each method from the class in these blanks

Automobile Example

- An *Automobile* class has the following methods
  - changeTires(Automobile a, Tire[] t)
  - checkOil(Automobile a)
  - drive(Automobile a)
  - getOilLevel()
  - start()
  - stop()
  - wash(Automobile a)
- Use the SRP analysis table to dismember it

SRP Analysis Table

- Fill the blanks with all the methods of *Automobile* class, and see if they can make sense for staying there

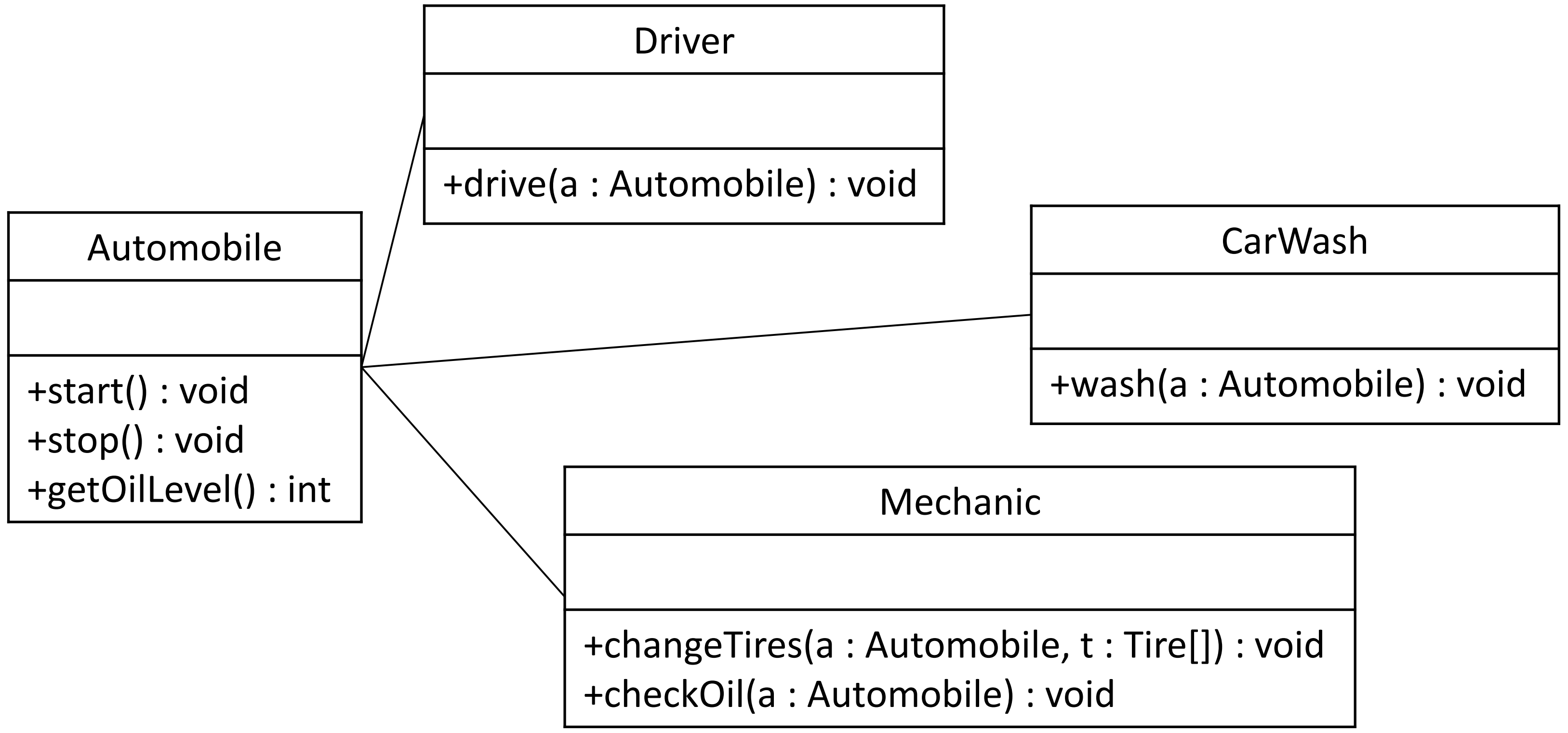
SRP Analysis for Automobile .				
The	Automobile		start	itself.
The	Automobile		stop	itself.
The	Automobile		changeTires	itself.
The	Automobile		drive	itself.
The	Automobile		wash	itself.
The	Automobile		checkOil	itself.
The	Automobile		getOilLevel	itself.

SRP	Not SRP
☑	☐
☑	☐
☐	☑
☐	☑
☐	☑
☐	☑
☑	☐

- Continue to analyze the ***Not SRP*** methods on a new table (new class) until there is no more ***Not SRP*** methods

Dismembering Automobile

- There are only three methods left in the *Automobile*
- Other methods are grouped into other classes


Driver
+drive(a : Automobile) : void
Automobile
+start() : void
+stop() : void
+getOilLevel() : int
CarWash
+wash(a : Automobile) : void
Mechanic
+changeTires(a : Automobile, t : Tire[]) : void
+checkOil(a : Automobile) : void

User Management Example

- Many information have their own user management module to keep user data in database
- The typical create transaction may look something like this

```
public class CreateUserTrans {
  public CreateUserTrans(int id, String name, String password) {
    // Suppose the DBOject does all the database processes behind
    DBObject dbObj = new DBObject();
    dbObj.connectDatabase();
    dbObj.create(id, name, password);
    dbObj.disconnectDatabase();
    System.out.println("ID: " + id);
    System.out.println("Name: " + name);
    System.out.println("Password: " + password);
  }
}
```

Dismembering the Codes

- By looking closer to the *CreateUser* transaction, we can dismember it into classes by adopting the MVC design pattern
- MVC (Model-View-Control) design pattern
  - Data Model: Transactional Data
  - View: Presentation
  - Control: Business logic / control flow

Dismembering the Codes (cont.)

- *CreateUser* transaction has the business logic part
  - Database process: connect, create, disconnect
  - From *Line 4* to *Line 7*
- *CreateUser* transaction has the presentation part
  - Show the result
  - From *Line 8* to *Line 10*
- *CreateUser* transaction has the transactional data
  - int id,
  - String name, and
  - String password

Presentation

- Create a class for displaying the transactional data

```
public class DisplayTools {
  public static void showUserInfo(UserBean bean) {
    System.out.println("ID: " + bean.getId());
    System.out.println("Name: " + bean.getName());
    System.out.println("Password: " + bean.getPassword());
  }
}
```

Data Model

- We can use a JavaBean *UserBean* to model the transactional data because it can be reused in this application

```
public class UserBean {
  private int id;
  private String name, String password;

  public int getId() { return id; }
  public void setId(int id) { this.id = id; }

  public String getName() { return name; }
  public void setName(String name) { this.name = name; }

  public String getPassword() { return password; }
  public void setPassword(String password) { this.password = password; }
}
```

Business Logic

- Create a class for database processes

```
public class DBProcess {
  public static void createUser(UserBean bean) {
    DBObject dbObj = new DBObject();
    dbObj.connectDatabase();
    dbObj.create(bean.getId(), bean.getName(), bean.getPassword());
    dbObj.disconnectDatabase();
  }
}
```

Control Flow

- The new *CreateUser* transaction become more flexible
- It will focus more on controlling the flow of a transaction and the use of objects

```
public class CreateUser {
  public CreateUser(int id, String name, String password) {
    UserBean bean = new UserBean();
    bean.setId(id);
    bean.setName(name);
    bean.setPassword(password);
    DBProcess.createUser(bean);
    DisplayTools.showUserInfo(bean);
  }
}
```

Delete User Transaction

- It is easier to make new transactions
- The *DeleteUser* transaction may look like this

```
public class DeleteUser {
  public DeleteUser(int id, String name, String password) {
    UserBean bean = new UserBean();
    bean.setId(id); // primary key of the table
    DBProcess.deleteUser(bean);
    DisplayTools.showUserInfo(bean);
  }
}
```

- Enhancing the *DBProcess* and *DisplayTools* classes do not require to change the (*CreateUser*, *DeleteUser*, *UserBean*, etc.) programs

Business Logic

- Enhance the *DBProcess* class

```
public class DBProcess {
  public static void createUser(UserBean bean) {
    DBObject dbObj = new DBObject();
    dbObj.connectDatabase();
    dbObj.create(bean.getId(), bean.getName(), bean.getPassword());
    dbObj.disconnectDatabase();
  }

  public static void deleteUser(UserBean bean) {
    DBObject dbObj = new DBObject();
    dbObj.connectDatabase();
    dbObj.delete(bean.getId());
    dbObj.disconnectDatabase();
  }
}
```

More about SRP

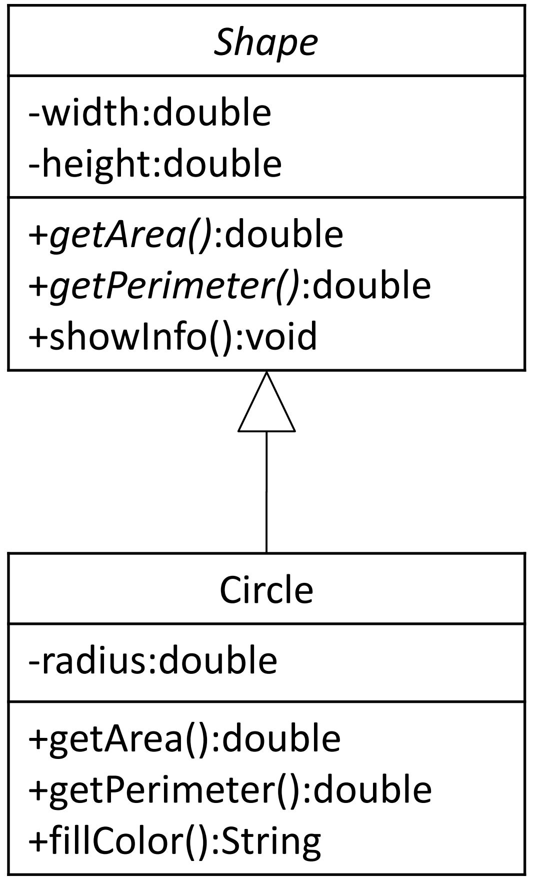
- This example dismembers a small program to four pieces. If the system only has one transaction, apply the SRP may not be efficient and valuable
- However, as the system grows large with more transactions, SRP can provide better management and easier for developers to expand the system
- SRP will often combine DRY and other principles together for building large systems
- One class does one thing or one type of thing, and no other classes share that behavior

Open-Closed Principle

- OCP is about allowing change but doing it without requiring to modify the existing codes
- Open for extension
  - The methods inside the super class can be overridden by subclasses and they are opened for extension
- Close for modification
  - The contents of super class (base class) is closed for modification. Subclasses cannot define new methods and their behaviors inside the super class

OCP Examples

- Superclass (*Shape*) has three methods opened for *extension*
  - public abstract double getArea();
  - public abstract double getPerimeter();
  - public void showInfo() {…}
- Subclass can override but cannot *modify* the contents of the methods inside the super class
- Subclass (*Circle*) cannot promote a method to super class
  - public String fillColor() {…}



More about OCP

- Abstracting the behavior away into a base class (or superclass) and locking them up from ***modification***
- Once a class is made, we don't want users to make changes. With OCP, we allow users to change the behaviors through ***extension***
- The principle is about flexibility and not just the inheritance
- There are many ways to apply OCP
  - methods in an *interface* (open for extension),
  - private methods in a class (closed for modification),
  - etc.

OCP Examples

- Calculate the monthly payment for the mortgage
- The formula is closed for further modification

```
import java.text.DecimalFormat;

public class MonthlyPayment {
  // close for modification
  private double getPayment(double loan, double rate, int year) {
    double interest = rate * 0.01 / 12;
    double tTerm = 1 + interest;
    double fTerm = tTerm;
    for (int i = 0; i < year * 12; i++) {
      fTerm *= tTerm;
    }
   return fTerm * loan;
 }
```

OCP Examples (cont.)

```
    public String calculate(double loan, double rate, int year) {
      System.out.println("Calculate the monthly payment");
      DecimalFormat df = new DecimalFormat("#,###,#00.00");
      return df.format(getPayment(loan, rate, year));
    }
    public static void main(String[] args) {
      MonthlyPayment mPay = new MonthlyPayment();
      System.out.println(mPay.calculate(1000000, 5, 20));
    }
  }
```

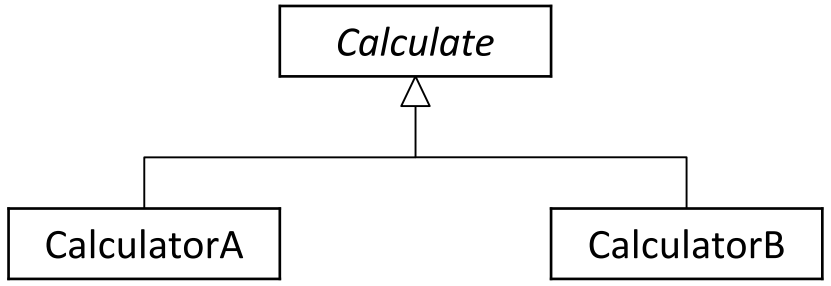
- Interface is opened for extension

```
  public interface BankCalculator {
    public double monthlyPayment(double loan, double rate, double time);
    public double maxLoanAmount(double mPay, double rate, double time);
    public double fixedDeposit(double amount, double rate, double time);
  }
```

Liskov Substitution Principle

- LSP was introduced by Barbara Liskov at 1987
- Subtypes must be substitutable for their base types
- We can use the subtype *calA* variable to replace the *myCal* variable and still **behave** the same

```
Calculate myCal;
myCal = new CalculatorA();
myCal.average(3,5);
myCal = new CalculatorB();
myCal.average(3,5);
…
CalculatorA calA;
calA = new CalculatorA();
calA.average(3,5);
```



LSP Example

- *CalculatorA* implements the *sum* method correctly and follows the LSP, and it is substitutable

```
public abstract class Calculate {
  public abstract double sum(double a, double b);
  public double average(double a, double b) {
    return sum(a, b) / 2;
  }
}
```

```
public class CalculatorA extends Calculate {
  public double sum(double a, double b) {
    return a + b;
  }
}
```

LSP Example (cont.)

- *CalculatorB* implements the *sum* method wrongly and cause the substitution to behave differently

```
public class CalculatorB extends Calculate {
  public double sum(double a, double b) {
    return a * b;
  }
}
```

It isn’t behaved as *SUM,* it violates the LSP

```
Calculate myCal;
myCal = new CalculatorA();
myCal.average(3, 5);
myCal = new CalculatorB();
myCal.average(3, 5);
```

They will not behave the same anymore

Interface Segregation Principle

- A client should never be forced to implement an interface that it doesn’t use or clients shouldn’t be forced to depend on methods they do not use
- ISP splits large interfaces into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them

High Cohesion

- ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy
- ISP is one of the five SOLID principles of object-oriented design, similar to the High Cohesion Principle of GRASP
- Low Cohesion interfaces contain many behaviors that clients will not use

ISP Example

- Define an interface with too many methods or the methods are not closely related
- An Animal class contains three behaviors (run, fly, and swim). However, most of the animals will not have all these abilities

```
public interface Animal {
  public void swim();
  public void run();
  public void fly();
}
```

ISP Example (cont.)

- In general, a bird can only fly but it cannot swim and run

```
public class Bird implements Animal {
  public void swim() { // useless method }
  public void run() { // useless method }
  public void fly() { System.out.println("FLY with wings"); }
}
```

- In general, a fish cannot run and fly

```
public class Fish implements Animal {
  public void swim() { System.out.println("Swim with fins"); }
  public void run() { // useless method }
  public void fly() { // useless method }
}
```

Adopt the ISP

- Decompose the Animal interface into different types (SeaAnimal, LandAnimal, and SkyAnimal)

```
public interface SeaAnimal {
  public void swim();
}

public interface LandAnimal {
  public void run();
}

public interface SkyAnimal {
  public void fly();
}
```

Adopt the ISP (cont.)

- Bird and Fish classes are more reasonable now

```
public class Bird implements SkyAnimal {
  public void fly() { System.out.println("FLY with wings"); }
}
```

```
public class Fish implements SeaAnimal {
  public void swim() { System.out.println("Swim with fins"); }
}
```

- Some species have two abilities

```
public class FlyingFish implements SeaAnimal, SkyAnimal {
  public void fly() { System.out.println("FLY with wings"); }
  public void swim() { System.out.println("Swim with fins"); }
}
```

Dependency Inversion Principle

- Martin Fowler introduced this concept at 1988
- Required object is not known at compile time but in runtime
- The binding process is achieved through Dependency Injection (DI)
- Follows the Dependency Inversion Principle
  - High-level modules should not depend on low-level modules, so they can reuse with different low-level modules
  - Instead of working with concrete classes, it will work with abstractions like interfaces or abstract classes
  - Let your application decide which concrete classes to the end uses
  - Create dependency between components in the modules
- Based on Hollywood Principle
  - Don’t call us, we’ll call you!

What is Hollywood Principle?

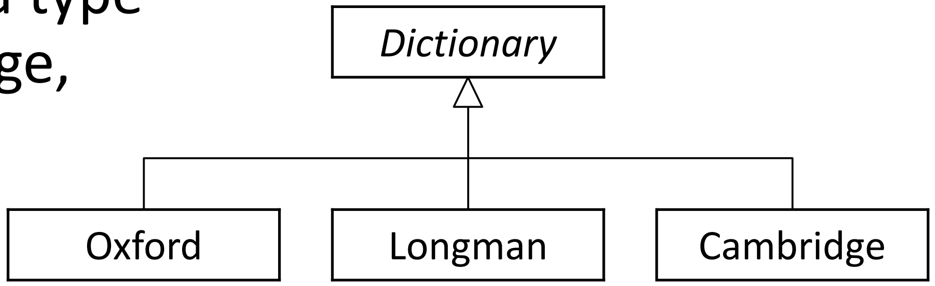
- "Don't call us, we will call you"
- A spell check application requires a dictionary component. The program may look like this in conventional programming style

```
public class SpellCheckApp {
  private OxfordDictionary oxford;
  public SpellCheckApp() {
    // Using the Oxford dictionary (Concrete Class)
    oxford = new OxfordDictionary();
  }
}
```

- Each object knows at compile time which are the real classes of the objects they need to interact with, and they will call them directly
- Your program has statically assigned a dictionary in compile time
- Programmers decided what users can use
- The program and object are the one calling Hollywood

Inversion Approach

- Instead of calling the Hollywood, here is to create dependency between spell check and dictionary
- Create a super class (*Dictionary*) as a type for all dictionaries (Oxford, Cambridge, Longman, etc.)
- The code will look like the following in IoC approach



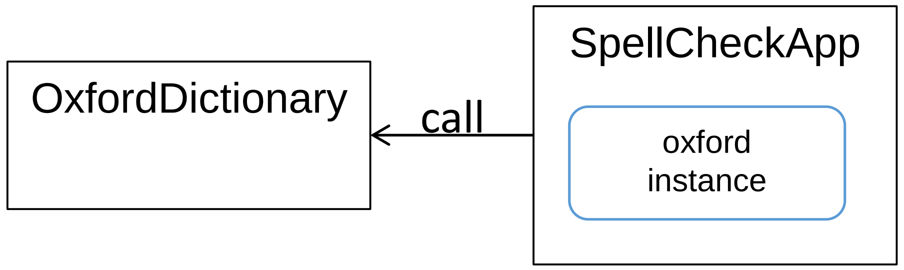
```
public class SpellCheck {
  private Dictionary dictionary; // super class (Dictionary type)
  public SpellCheck(Dictionary dictionary) {
    // Using an unknown dictionary
    this.dictionary = dictionary;
  }
}
```

Inversion Approach (cont.)

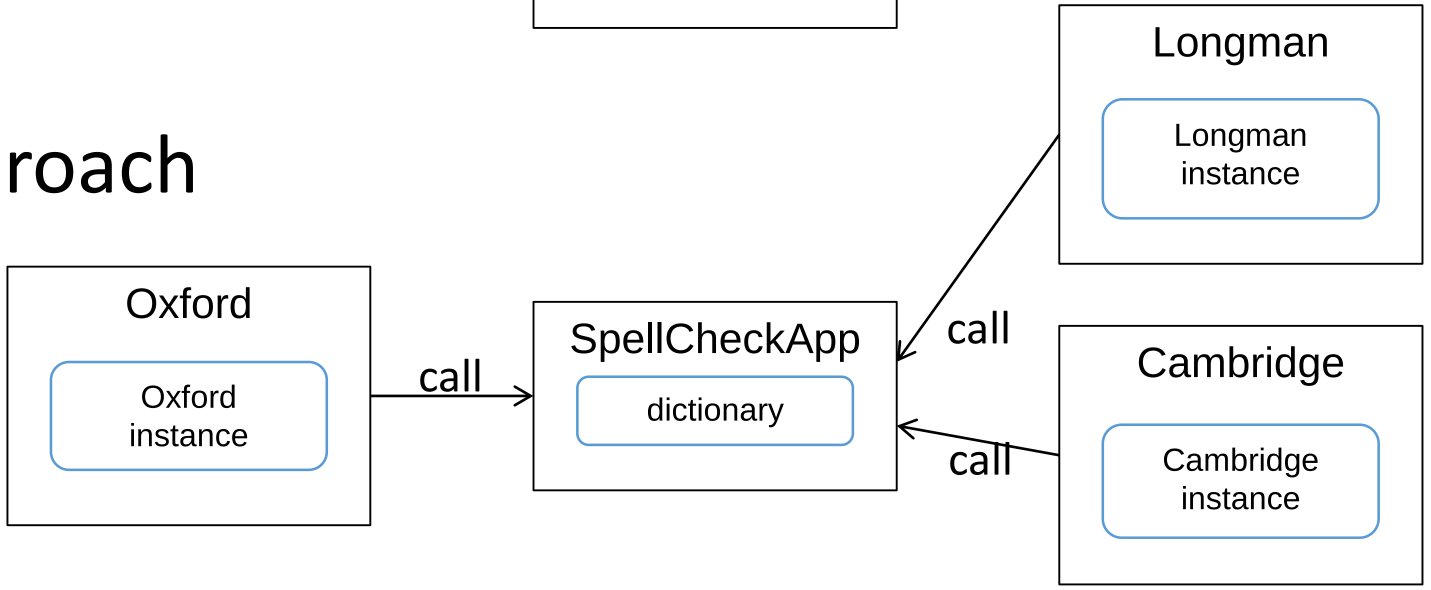
- The program won’t create an instance of an object (dictionary) but use the object from callers at run time
- Now, the caller can flexibly decide whatever dictionary (Oxford, Cambridge, Longman, etc.) to be used when calling this spell check application
- The program doesn’t know which dictionary will be used at compile time but will know it at runtime
- The program waits for callers (Hollywood) to make a call

IoC Concept

- Conventional Approach



- IoC Approach



Word Processor without IoC

```
public class WordProcessor1 {
  private OxfordDictionary oxford;
  private String paragraph;

  public WordProcessor1(String paragraph) {
    // Using the concrete class Oxford dictionary
    oxford = new OxfordDictionary();
    this.paragraph = paragraph;
  }

  public String spellCheck() {
    return oxford.checkSpell(this.paragraph);
  }
}

```

Word Processor with IoC (setter)

```
public class WordProcessor2 {
  private Dictionary dictionary;
  private String paragraph;

  public void setDictionary(Dictionary dictionary) {
    // Setter method to define what dictionary to be used
    this.dictionary = dictionary;
  }

  public void setParagraphy(String paragraph) {
    this.paragraph = paragraph;
  }

  public String spellCheck() {
    return dictionary.checkSpell(this.paragraph);
  }
}
```

Word Processor with IoC (constructor)

```
public class WordProcessor3 {
  private Dictionary dictionary;
  private String paragraph;

  public WordProcessor3(Dictionary dictionary) {
    // Define what dictionary to be used on creating the WordProcessor
    this.dictionary = dictionary;
  }

  public void setParagraphy(String paragraph) {
    this.paragraph = paragraph;
  }

  public String spellCheck() {
    return dictionary.checkSpell(this.paragraph);
  }
}
```

Word Processors Implementation

```
public class TestWordProcessor {
  public static void main(String arg[]) {
    String paragraph = "Testing the IoC program.";
    Dictionary longman = new Longman(); // dictionary subclass
    Dictionary cambridge = new Cambridge(); // dictionary subclass
    // Conventional style
    WordProcessor1 wp1 = new WordProcessor1(paragraph);
    // Setter Injection
    WordProcessor2 wp2 = new WordProcessor2();
    wp2.setParagraph(paragraph);
    wp2.setDictionary(longman);
    // Construction Injection
    WordProcessor3 wp3 = new WordProcessor3(cambridge);
    wp3.setParagraph(paragraph);
    System.out.println(wp1.spellCheck() + wp2.spellCheck() + wp3.spellCheck());
  }
}
```

Constructor vs Setter Injection

- Constructor Injection
  - Has no setter methods, so it cannot reconfigure the properties
  - Ensure that the properties have initial values or instances
  - Partial injection of dependencies can possible in setter injection
  - Objects in setter injection is mutable where constructor injection is immutable
- Setter injection
  - Able to change the properties after creation
  - Possible to create the objects without calling the setter method (Lazy loading). This could leave properties in an uninitialized state
- In the word processor example, constructor injection is more suitable because it requires to create the initial dictionary instance first

Summary

- Design principles help us to make abstraction and decomposition in more reasonable ways
- By following the design principles, we can write codes not just more elegant and readable but we can develop complicate enterprise level systems faster and easier, and they can be more maintainable, flexible, or extensible