Software Development

Dr. Hamada I. AbdulWakel[1]

[1]Computer Science Department

2022 - 2023

Ch3: Code Construction Lec#5

Code Refactoring

Definition

A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its external behavior.

How to write great code that is easy to modify and understand. Making code better without changing the functionality.

Code Refactoring ...

- Reasons:
  1. Bad code quality.
  2. Code smells.
  3. Technical debt.
  4. Complex code.
- Advantages:
  1. Provides more readable and modular code.
  2. Improves software extendibility.
  3. Reduce the expense of code maintenance.
  4. Architecture improvement without impacting software behavior.
- Challenges:
  1. Time constraint.
  2. Reluctance.
  3. Fear factor.
  4. Re-Testing.

Code Smell

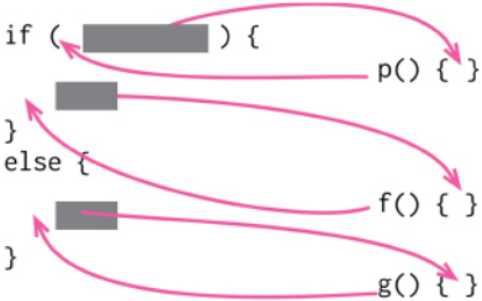
1. **Duplicated Code:** identical or very similar code exists in more than one location.
2. **Long Method:** a method that has grown too large.
3. **Long Class:** a class that has grown too large.
4. **Too Many Parameters:** a long list of parameters is hard to read and makes calling and testing the function complicated.
5. **Feature Envy:** a class that uses methods of another class excessively.
6. **Lazy Class:** a class that does too little.
7. **Contrived Complexity:** forced usage of overly complicated design patterns where the simpler design would suffice.
8. **Complex to debug:** code has become complex enough to debug.

Refactoring Techniques

1. Composing methods.
2. Making method calls simpler.
3. Moving features within objects.
4. Organizing data.
5. **Simplifying conditional expression.**

Tech1: Decompose Conditional

- **Problem:** Code has a complex conditional (if-then/else or switch).
- **Solution:** Decompose the complicated parts of the conditional into separate methods.



Tech1: Decompose Conditional Example

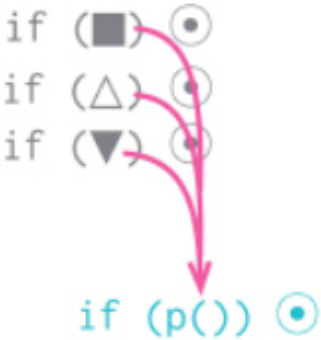
```
if ( date.before(SUMMER_START) || date.after(SUMMER_END) )
  charge = quantity * winterRate + winterServiceCharge;
else
  charge = quantity * summerRate;

if (isSummer(date))
  charge = summerCharge(quantity);
else
  charge = winterCharge(quantity);

public boolean isSummer(date){
  return date.before(SUMMER_START) || date.after(SUMMER_END);
}
```

Tech2: Consolidate Conditional Expression

- **Problem:** Code has multiple conditionals that lead to the same action.
- **Solution:** Consolidate all these conditionals in a single expression.



Tech2: Consolidate Conditional Expression Example

```
double disabilityAmount() {
  if (seniority < 2) return 0;
  if (monthsDisabled > 12) return 0;
  if (isPartTime) return 0;
  // Compute the disability amount.
}

double disabilityAmount() {
  if (isNotEligibleForDisability()) return 0;
  // Compute the disability amount.
}
```

Tech3: Replace Nested Conditional with Guard Clauses

- **Problem:** Code has a group of nested conditionals and it's hard to determine the normal flow of code execution.
- **Solution:** Isolate all special checks into separate clauses ("flat" list of conditionals) and place them before the main checks.

Tech3: Replace Nested Conditional with Guard Clauses Example

```
public double getPayAmount() {
  double result;
  if (isDead) result = deadAmount();
  else {
    if (isSeparated) result = separatedAmount();
    else {
      if (isRetired) result = retiredAmount();
      else result = normalPayAmount();
    }
  }
  return result;
}
```

Tech3: Replace Nested Conditional with Guard Clauses Example ...

```
public double getPayAmount() {
  if (isDead) return deadAmount();
  if (isSeparated) return separatedAmount();
  if (isRetired) return retiredAmount();
  return normalPayAmount();
}
```

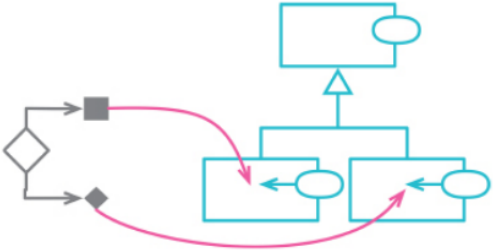
Tech3: Replace Nested Conditional with Guard Clauses Example ...

```
private dataType doSomething(){
  if (everythingIsGood()) return Value;
  else return otherValue;
}

private dataType doSomething(){
  if ( ! everythingIsGood()) return otherValue;
  return otherValue;
}
```

Tech4: Replace Conditional with Polymorphism

- **Problem:** Code has a conditional that performs various actions depending on object type or properties.
- **Solution:** Create subclasses matching these branches. In them, create a shared method and move code from the corresponding branch of the conditional to it.



Tech4: Replace Conditional with Polymorphism Example

```
class Bird {
  double getSpeed() {
    switch (type) {
      case EUROPEAN: return getBaseSpeed(); break;
      case AFRICAN:
        return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
        break;
      case NORWEGIAN_BLUE:
        return (isNailed) ? 0 : getBaseSpeed(voltage); break;
    }
    throw new RuntimeException("Should be unreachable");
  }
}
```

Tech4: Replace Conditional with Polymorphism Example ...

```
abstract class Bird { abstract double getSpeed(); }
class European extends Bird {
  double getSpeed() { return getBaseSpeed(); }
}
class NorwegianBlue extends Bird {
  double getSpeed() {
    return (isNailed) ? 0 : getBaseSpeed(voltage);
  }
}
// Somewhere in client code
speed = bird.getSpeed();
```

Ch4: Solid Principles

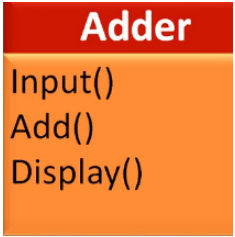
Code Rot

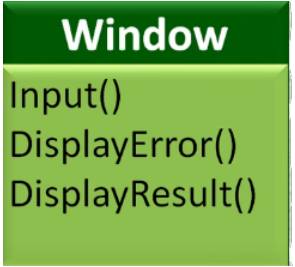
- The degradation of the code base over time.
- Reasons:
  1. Rigidity.
  2. Fragility.
  3. Immobility.
- Results:
  1. Inflexibility.
  2. Hard to maintain.
  3. Resilience.

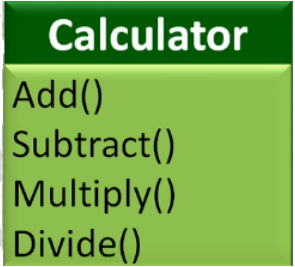
Cohesion & Coupling in OOP

- **Cohesion:** the class is fully self-contained and that every single method in it utilizes every instance variable declared within the class.

Cohesion Example







Cohesion & Coupling in OOP ...

- **Coupling:** the state of a system where a change in one class does not affect any other classes.

High Coupling Example

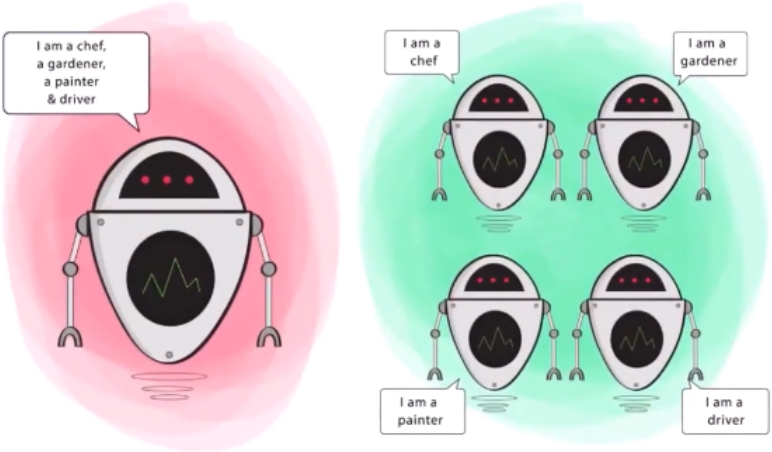
```
public class A{
    B b;
    ⋮
}
```

```
public class B{
    ⋮
}
```

SOLID Principles

- Single Responsibility Principle.
  “A class should have one, and only one, reason to change”.
- Open Closed Principle.
  "You should be able to extend a class’s behavior, without modifying it".
- Liskov Substitution Principle.
  “Derived classes must be substitutable for their base classes”.
- Interface Segregation Principle.
  “Make fine grained interfaces that are client specific”.
- Dependency Inversion Principle.
  “Depend on abstrations, not on concretions”.

Single Responsibility Principle (SRP)



Single Responsibility Principle (SRP) ...

Problem

```
public class Customer{
  private String name;
  void storeCustomer(String customerName){ ....... }
  void generateCustomerReport(String customerName){ ........ }
}
```

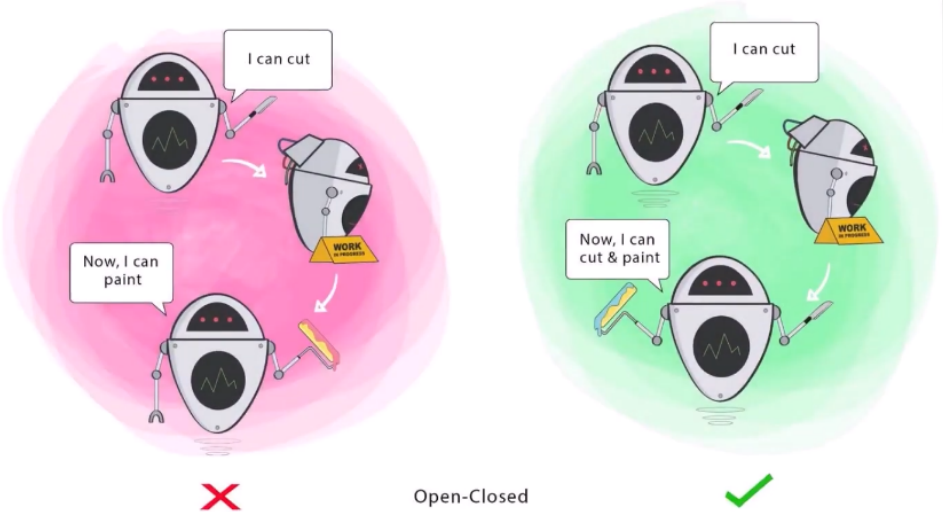
Solution

```
public class Customer{ ....... }
public class CustomerDB{ ....... }
public class CustomerReportGenerator{ ....... }
```

Single Responsibility Principle (SRP) ...

- SRP states that every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class, module or function.
- Important for testing, lower coupling, and easier to understand and organize code.
- Achieving this principle by grouping together things that change for the same reason.
- Violating this principle can be determined by identifying things that are changing for different reasons.

Open Closed Principle (OCP)



Open Closed Principle (OCP) ...

Problem

```
public class Rectangle{
   private int width;
   private int height;
}
public class Square{
   private int side;
}
public class ShapePrinter{
   public void drawShape(Object shape){
      if (shape instanceof Rectangle){    // draw Rectangle    }
      else if (shape instanceof Square){    // draw Rectangle    }
   }
}
```

Open Closed Principle (OCP) ...

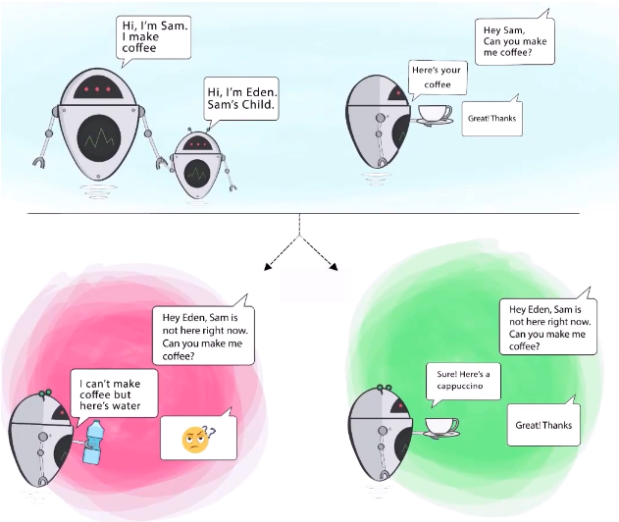
Solution

```
public abstract class Shape{    abstract void draw();    }
public class Rectangle extends Shape{
   private int width, height;
   @override
   public void draw(){ ..... }
}
public class ShapePrinter{
   public void drawShape(Shape shape){
     shape.draw();
   }
}
```

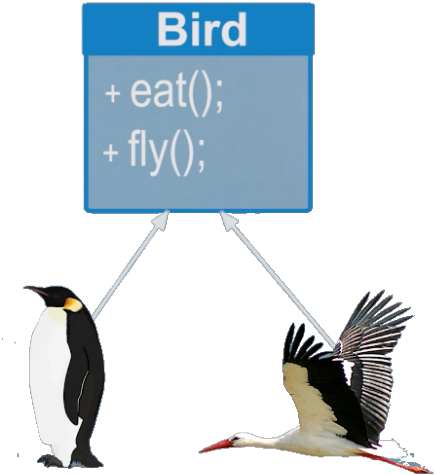
Open Closed Principle (OCP) ...

- Any software entity (function, class, module) should be open for extension but closed for modification.
- Abstraction and Polymorphism are the keys!
- OCP can greatly enhance reusability and maintainability.
- Extensible systems are used for requirements change.

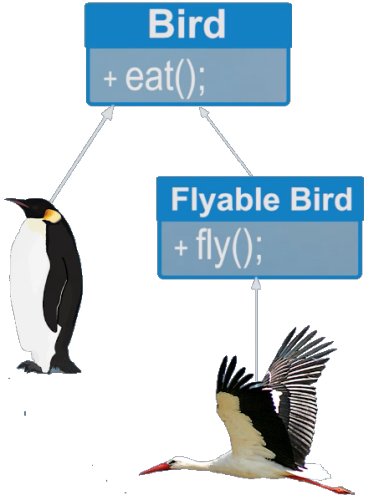
Liskov Substitution Principle (LSP)



Liskov Substitution Principle (LSP) ...



Liskov Substitution Principle (LSP) ...



Liskov Substitution Principle Example

Problem

```
abstract class Account{
  //attributes
  protected Account(string name, double balance){}
  // getters and setters
  public abstract void Deposit(double amount);
  public abstract void Withdraw(double amount);
}
class FixedDepositeAccount extends Account{
  // constructor call super.
  public void Deposit(double amount){ this.balance += amount; }
  public void Withdraw(double amount){
throw new RuntimeException("Withdraw not supported here !!");
  }
}
```

Liskov Substitution Principle Example ...

Problem ...

```
class Client {
  public static void main(String[] args) {
    Account facc = new FixedDepositeAccount("hamada", 10000);
    facc.Withdraw(1000);
  }
}
```

Liskov Substitution Principle Example

Solution

```
class Account{
  //attributes
  protected Account(string name, double balance){}
  // getters and setters
  public abstract void Deposit(double amount);
}
abstract class AccountwithWithdraw extends Account{
  // constructor call super.
  public abstract void Withdraw(double amount);
}
class FixedDepositeAccount extends Account{
  // constructor call super.
  public void Deposit(double amount){ this.balance += amount; }
}
```

Liskov Substitution Principle (LSP) ...

- **Substitutability** is a OOP principle stating that an object and a sub-object must be interchangeable without breaking the program.

- If S is a subtype of T, then objects of type T can be replaced with objects of type S.
  T t1 = new T();
  T t1 = new S(); //without any problem

- LSP is violated when adding a derived class requires modifications of the base class.