COMP2511

Refactoring

Prepared by

Ashesh Mahidadia

Refactoring: Motivation

- Code refactoring is the process of restructuring existing computer code without changing its external behavior.
- Originally Martin Fowler and Kent Beck defined refactoring as,

  "A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior… It is a disciplined way to clean up code that minimizes the chances of introducing bugs."
- **Advantages**: improved code readability, reduced complexity; improved maintenance and extensibility
- If done well, helps to identify *hidden* or *dormant* bugs or vulnerabilities, by simplifying code logic.
- If done poorly, may change external behavior, and/or introduce new bugs!
- Refactoring is different to adding features and debugging.

Refactoring: Motivation

- Refactoring is usually motivated by noticing a *code smell (possible bad design/coding practices)*.
- Code Smell is a hint that something might be wrong, **not** a certainty.
- Identifying a Code Smell allows us to re-check the implementation details and consider possible *better* alternatives.
- Automatic unit tests should be set up before refactoring to ensure routines still behave as expected.
- Refactoring is an iterative cycle of making a small program transformation, testing it to ensure correctness, and making another small transformation.

Software Maintenance

- Software Systems evolve over time to meet new requirements and features.
- Software maintenance involve:
  - Fix bugs
  - Improve performance
  - Improve design
  - Add features
- Majority of software maintenance is for the last three points!
- Harder to maintain code than write from scratch!
- Most of the development time is spent in maintenance!
- Good design, coding and planning can reduce maintenance pain and time!
- Avoid code smells to reduce maintenance pain and time!

Code Smells: Possible Indicators

- Duplicated code
- Poor abstraction (change one place → must change others)
- Large loop, method, class, parameter list; deeply nested loop
- Class has too little cohesion
- Modules have too much coupling
- Class has poor encapsulation
- A subclass doesn't use majority of inherited functionalities
- A "data class" has little functionality
- Dead code
- Design is unnecessarily general
- Design is too specific

Low-level refactoring

- Names:
  - Renaming (methods, variables)
  - Naming (extracting) “magic” constants
- Procedures:
  - Extracting code into a method
  - Extracting common functionality (including duplicate code) into a class/method/etc.
  - Changing method signatures
- Reordering:
  - Splitting one method into several to improve cohesion and readability (by reducing its size)
  - Putting statements that semantically belong together near each other

- For more, see **http://www.refactoring.com/catalog/**

IDEs support low-level refactoring

- Renaming:
  - Variable, method, class.
- Extraction:
  - Method, constant
  - Repetitive code snippets
  - Interface from a type
- Inlining: method, etc.
- Change method signature.
- Warnings about inconsistent code.

Higher-level refactoring

- Refactoring to design patterns.
- Changing language idioms (safety, brevity).
- Performance optimization.
- Generally high-level refactoring is much more important, but unfortunately not well-supported by tools.

Code Smells

Code and Design Smells

Smells : Design aspects that violate fundamental design principles and impact software quality

Smells occur at different levels of granularity

- Code Smells: Structures in implementation of code such as large methods, classes with multiple responsibilities, complex conditional statements that lead to poor code
- Design Smells: Design aspects at a higher level of abstraction (class level abstractions) such as classes with multiple responsibilities, refused bequest

Regardless of the granularity, smells in general indicate violation of software design principles, and eventually lead to code that is rigid, fragile and require “refactoring”

Smells

Bloaters: Code, Methods and classes that have grown in size, that they are hard to work with

- Long Method, Large Class, Long Parameter List, Data Clumps

OO Abusers: Result from incorrect or incomplete application of OO principles

- Switch statements, Refused Bequest

Change Preventers: Code changes are difficult (rigid code)

- Divergent change, Shot Gun Surgery

Dispensables: Code that is pointless and unnecessary

- Comments, Data Class, Lazy Class, Duplicate code

Couplers: Excessive coupling between classes

- Feature Envy, Inappropriate intimacy, Message Chains

Smell: Long Method

Fix smell, long method

- Reduce length of a method body via **Extract Method**
  - More readable, Less code duplication
  - Isolates independent parts of code, - errors are less likely
- If local variables and parameters interfere with extracting a method, use
  - **Replace Temp With Query**
  - **Introduce Parameter Object**
  - **Preserve Whole Object**
- If the above doesn't work, try moving the entire method to a separate object via **Replace Method with Method Object**
- **Replace Method with Method Object**
- Conditional operators and loops are a good clue that code can be moved to a separate method.

Refactoring Techniques – Extract Method

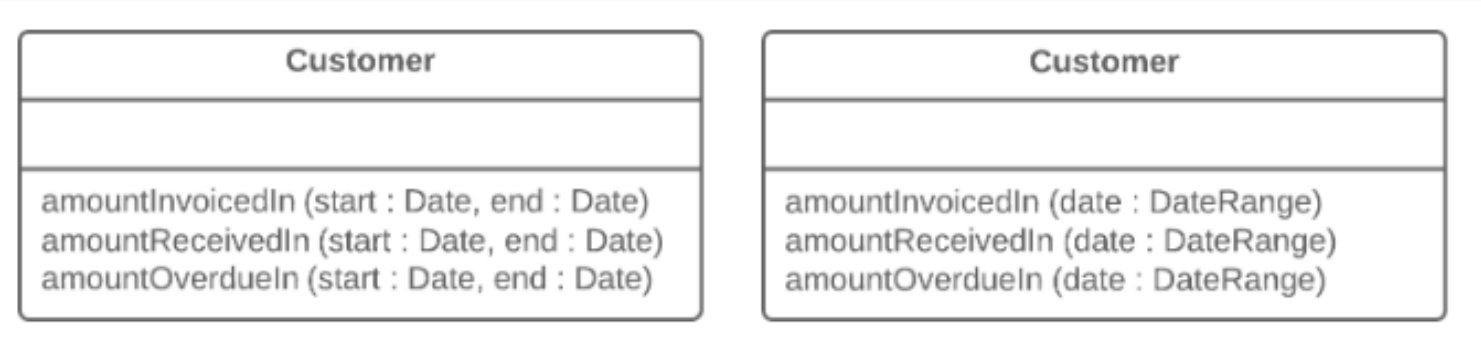
- More readable code (The new method name should describe the method's purpose)
- Less code duplication, more reusability
- Isolates independent parts of code, meaning that errors are less likely
- A very common refactoring technique for code smells

```
public void debit(float amount) {
    // deducts amount by balance
    balance -= amount;

    // records transaction
    transactions.add(new Transaction(amount, true));

    // record last transaction date
    lastTransactionDate = LocalDate.now().toString();
}
```

```
public void debit(float amount) {
    deductBalance(amount);
    recordTransaction(amount, true);
    recordLastTransaction();
}

private void deductBalance(float amount) {
    balance -= amount;
}
private void recordTransaction(float amount, boolean isDebit) {
    transactions.add(new Transaction(amount,isDebit));
}
private void recordLastTransaction() {
    lastTransactionDate = LocalDate.now().toString();
}
```

Refactoring Techniques: Introduce Parameter Object

- Methods contain a repeating group of parameters, causing code duplication
- Consolidate these parameters into a separate class
  - Also helps to move the methods for handling this data
  - Beware, if only data is moved to a new class and associated behaviours are not moved, this begins to smell of a **Data Class**



- Eliminates smell such as Long Parameter List, Data Clumps, Primitive Obsession, Long Method

Refactoring Technique: Replace Temp With Query

Often, we place the result of an expression in a local variable for later use in the code

With Replace Temp With Query we:

- Move the entire expression to a separate method and return the result from it.
- Query the method instead of using a variable
- Reuse the new method in other methods

```
double calculateTotal() {
 double basePrice = quantity * itemPrice;
if (basePrice > 1000) {
  return basePrice * 0.95;
}
else {
 return basePrice * 0.98;
}
}
```

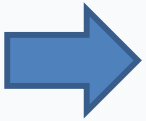
```
double calculateTotal() {
 if (basePrice() > 1000) {
       return basePrice() * 0.95; }
else {
    return basePrice() * 0.98; } }
double basePrice() {
  return quantity * itemPrice;
}
}
```

- Eliminates smell such as Long Method, Duplicate Code

Refactoring Technique: Extract Class

- Having all the phone details in class Customer is not a good OO design and also breaks SRP
- Refactor into two separate classes, each with its appropriate responsibility

```
public class Customer {
    private String name;
    private String workPhoneAreaCode;
    private String workPhoneNumber;
    private String homePhoneAreaCode;
    private String homePhoneNumber;
}
```



```
public class Customer {
    private String name;
    private Phone workPhone;
    private Phone homePhone;

}
```

```
public class Phone {
    public String areaCode;
    public String phoneNumber;

}
```

Smell: Large Class

- Problem:
- Similar to Long Method
- Usually violates Single Responsibility Principle
- May have
  - A large number of instance variables
  - Several methods
- Typically lacks cohesion and potential for duplicate code smell

Solution:

- Bundle group of variables via **Extract Class** or **Extract Sub-Class**

Code Smell: Long Parameter List

Problem: Calling a query method and passing its results as the parameters of another method, while that method could call the query directly

- Too many parameters to remember
- Bad for readability, usability and maintenance

```
public String getSummary() {
    return buildCustomerSummary(getFirstName(), getLastName(), getTitle(),
            address.getCity(), address.getPostCode());
}

private String buildCustomerSummary(String firstName, String lastName,
        String title, String city, String postCode) {
    return title + " " + firstName + " " + lastName + "," + city + "," + "postcode";
}
```

Solution:

- try placing a query call inside the method body via **replace**

```
// 1. Apply replace parameter with method call
// 2. Apply change method signature to remove the first three parameters
// 3. Preserve whole object - passing in the entire object instead of object data
private String buildCustomerSummary(Address address) {
    return getTitle() + " " + getFirstName() + " " + getLastName() + "," + address.getCity() + ","
            + address.getPostCode();
}
```

Code Smell: Data Clumps

Problem:

- Different parts of the code contain identical groups of variables e.g., fields in many classes, parameters in many method signatures
- Can lead to code smell Long Parameter List

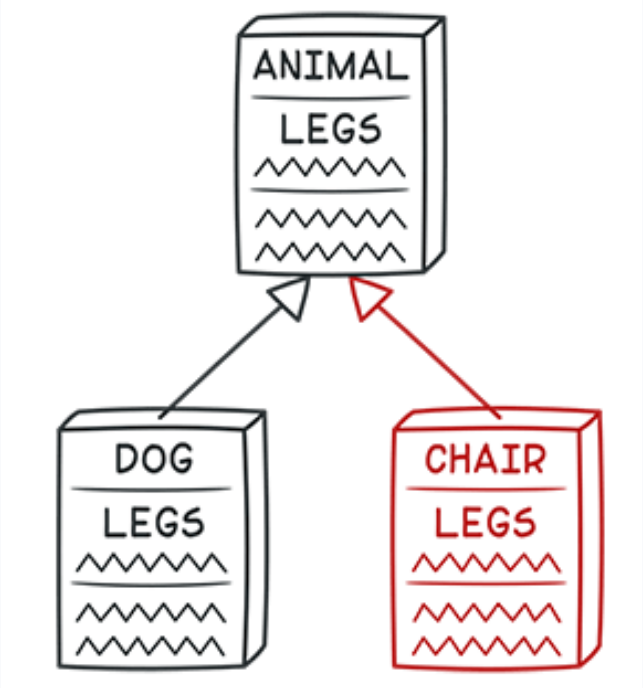
Solution: Move the behaviour to the data class via **Move Method**

- If repeating data comprises the fields of a class, use **Extract Class** to move the fields to their own class.
- If the same data clumps are passed in the parameters of methods, use **Introduce Parameter Object** to set them off as a class.
- If some of the data is passed to other methods, think about passing the entire data object to the method instead of just individual fields **Preserve Whole Object** will help with this.

Code Smell: Refused Bequest

Problem:

- A subclass uses only some of the methods and properties inherited from its parents
- The unneeded methods may simply go unused or be redefined and give off exceptions
- Often caused by creating inheritance between classes only by the desire to reuse the code in a super-class



Code Smell: Refused Bequest

Solution:

- If inheritance makes no sense and the subclass really does have nothing in common with the superclass, eliminate inheritance in favour of **Replace Inheritance with Delegation**
- If inheritance is appropriate, but super class contains fields and methods not applicable to all classes, then consider the following options
  - Create a new subclass
  - Apply **Push Down Field** to move field relevant only to subclass from superclass
  - Apply **Push Down Method** to move behaviour from super class to sub class, as behaviour makes sense only to sub class
  - Often, you may apply an **Extract Sub-Class Class** to combine the above steps

Refused Bequest Example

class Camel does not use field model. It should be pushed down to class Car

```
public abstract class Transport {
  // Push Down Field
  private String model;
  // Push Down Method
  public String getModel() throws Exception
  {
        return model;
  }
  ...
}
public class Car extends Transport { ... }
public class Camel extends Transport {
  ...
  public String getModel() {
        throw new NotSupportedException();
  }
}
public abstract class Transport {
    ...
}
```

```
// Use Push Down Field to move field and
// Push Down Method to move behaviour
// only relevant to sub class
// from super class to sub class

public abstract class Transport {
        ...
}
public class Car extends Transport
{
  private String model;
  public String getModel()
  {
        return model;
  }
  ...
}
public class Camel extends Transport
{
  ...
}
```

Code Smell: Duplicate Code

Code Fragments look similar

- If the same code is found in two or more methods in the same class: use **Extract Method** and place calls for the new method in both places
- If the same code is found in two subclasses of the same level:
  - Use **Extract Method** for both classes, followed by **Pull Up Field** for the fields used in the method that you are pulling up.
  - If the duplicate code is inside a constructor, use **Pull Up Constructor Body**
  - If the duplicate code is similar but not completely identical, use **Form Template Method**
  - If two methods do the same thing but use different algorithms, select the best algorithm and apply **Substitute Algorithm**
- If duplicate code is found in two different classes:
  - If the classes are not part of a hierarchy, use **Extract SuperClass** in order to create a single superclass for these classes that maintains all the previous functionality

Code Smell: Feature Envy

Problem: A method that is more interested in a class other than the one it actually is

- Invokes several methods on another object to calculate some value
- Creates unnecessary coupling between the classes

Solution: A goal of OO design is to put the methods with its associated data

- So the method must moved to the relevant class via **Move Method**
- If only part of a method accesses the data of another object, use **Extract Method** followed by **Move Method** to move the part in question
- If a method uses functions from several other classes, first determine which class contains most of the data used. Then place the method in this class along with the other data.

Code Smell: Divergent Change, Shot Gun Surgery

Divergent Change: One class is changed in different ways for different reasons

- **Solution:** Any change to handle a variation should change a single class, and all the typing in the new class should express the variation.
- To clean this up you identify everything that changes for a particular cause and use Extract Class to put them all together

Shot Gun Surgery: A small change in the code forces lots of little changes to different classes

- **Solution:**
  - Use **Move Method** or **Move Field** to put all the changes into a single class
  - Often you can use **Inline Class** to bring a whole bunch of behaviour together.
- Divergent change is one class that suffers many kinds of changes, and shotgun surgery is one change that alters many classes.

Code Smell: Data Classes

Problem: Classes that just have attributes with setters and getters and no behaviour

One of the goals of OO design is to put behaviour where the data is

```
public class CustomerSummaryView {
    private Customer customer;
    public CustomerSummaryView(Customer customer) {
        this.customer = customer;
    }
    public String getCustomerSummary() {
        Address addr = customer.getAddress();
        return customer.getTitle() + " " + customer.getFirstName() + " "
        + customer.getLastName() + "," + addr.getCity() + "," + addr.getPostCode();
    }
}
```

Solution: Move the behaviour to the data class via **Move Method**

```
// 1. Apply move method inside method getCustomerSummary() in address class
//    to move method to class Customer
// 2. Extract the address summary
// 3. Move the address summary to the class Address

public String getCustomerSummary() {
    return getTitle() + " " + getFirstName() + " "
    + getLastName() + "," + address.getAddressSummary();
}
```

Code Smell: Lazy classes

Problem: Classes that aren't doing much to justify their existence (maintenance overhead)

Subclasses without any overridden methods or additional fields can be lazy classes as well

```
public class PostCode {
    private String postcode;

    public PostCode(String postcode) {
        this.postcode = postcode;
    }

    public String getPostcode() {
        return postcode;
    }

    public String getPostcodeArea(){
        return postcode.split("")[0];
    }
}
```

```
public class Address {

    private final String number;
    private final String street;
    private final String city;
    private final String country;
    private final PostCode postcode;

    public Address(String no, String st, String city,
                   PostCode pCode, String country) {
        this.number = no;
        this.street = st;
        this.city = city;
        this.postcode = pCode;
        this.country = country;
    }
```

Solution:

- Move the data (postcode) from lazy class PostCode to the class Address
- Delete the lazy class

Code Smell: Switch Statements

Problem: Switch statements are bad from an OO design point of view

```
public RiskFactor calculateMotoristRisk() {
    if (motorist.getPointsOnLicense() > 3 || motorist.getAge() < 25) {
        return RiskFactor.HIGH;
    }
    if (motorist.getPointsOnLicense() > 0 ) {
        return RiskFactor.MEDIUM;
    }
    return RiskFactor.LOW;
}

public double calculateInsurancePremium(double insuranceValue) {
    RiskFactor riskFactor = calculateMotoristRisk();
    switch(riskFactor) {
        case LOW:
            return insuranceValue * 0.02;
        case MEDIUM:
            return insuranceValue * 0.04;
        default:
            return insuranceValue * 0.06;
    }
}
```

Solution: Replace switch statements with a polymorphic solution based on Strategy Pattern applying a series of refactoring techniques *(Extract Method, Move Method, Extract Interface etc., Refer lecture demo for complete solution)*

List of Refactoring Techniques to be familiar

Move Field/Method

Extract Class/Inline Class

Extract Method

Inline Method/Temporary Variable

Replace Temp with Query

Replace Method with Method Object

Rename Method

Substitute Algorithm

Introduce Parameter Object

Preserve Whole Object

Extract Sub Class/Super Class/Interface

Extract Method

Pull Up Field/Method/Constructor Body

Form Template Method

Replace Inheritance with Delegation

Replace Conditional with Polymorphism

Useful Links

https://refactoring.guru/refactoring/smells

https://www.refactoring.com/catalog/