Agile Software Development

Lecturer: Raman Ramsin

Lecture 13

Refactoring – Part 1
Refactoring: Definition

**Refactoring:**

- A change made to the internal structure of software to make it
  - easier to understand, and
  - cheaper to modify.

- *The observable behavior of the software should not be changed.*
Refactoring: Why?

Why Should You Refactor?

- Refactoring Improves the Design of Software
- Refactoring Makes Software Easier to Understand
- Refactoring Helps You Find Bugs
- Refactoring Helps You Program Faster
Refactoring: When?

When Should You Refactor?

- Refactor the third time you do something similar (The Rule of Three)
- Refactor When You Add Function
- Refactor When You Need to Fix a Bug
- Refactor As You Do a Code Review
Symptoms of Bad Code (1)

1. Mysterious Name
2. Duplicated Code
3. Long Function
4. Long Parameter List
5. Global Data
6. Mutable Data
7. **Divergent Change:** When one class is commonly changed in different ways for different reasons.
8. **Shotgun Surgery:** When every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.
Symptoms of Bad Code (2)

9. **Feature Envy:** A method that seems more interested in a class other than the one it actually is in.

10. **Data Clumps:** Bunches of data that regularly appear together.

11. **Primitive Obsession:** Excessive use of primitives, due to reluctance to use small objects for small tasks.

12. **Repeated Switches**

13. **Loops**

14. **Lazy Element:** An Element that isn't doing enough to justify its maintenance.

15. **Speculative Generality:** Classes and features have been added just because a need for them may arise someday.
Symptoms of Bad Code (3)

16. **Temporary Field:** An attribute that is set only in certain circumstances.

17. **Message Chains:** Transitive visibility chains.

18. **Middle Man:** Excessive delegation.

19. **Insider Trading:** Excessive interaction and coupling.

20. **Large Class**

21. **Alternative Classes with Different Interfaces**

22. **Data Class**

23. **Refused Bequest:** When children don’t fulfill their parents’ commitments.

24. **Comments:** When comments are used to compensate for bad code.
Refactoring Patterns: Categories

- **First Set:** The most commonly used refactorings
- **Encapsulation:** Enhancing information hiding
- **Moving Features:** Moving elements between contexts
- **Organizing Data:** Making data easier to work with
- **Simplifying Conditional Logic:** Making conditional logic less error-prone
- **Refactoring APIs:** Making interfaces easy to understand and use
- **Dealing with Inheritance:** Moving features around a hierarchy of inheritance
First Set: *Extract Function*

**Extract Function**
- You have a code fragment that can be grouped together.
- *Turn the fragment into a function whose name explains the purpose of the function.*

```java
void printOwing() {
    printBanner();

    //print details
    System.out.println("name: " + _name);
    System.out.println("amount " + getOutstanding());
}
```

```java
void printOwing() {
    printBanner();
    printDetails(getOutstanding());
}
```

```java
void printDetails (double outstanding) {
    System.out.println("name: " + _name);
    System.out.println("amount " + outstanding);
}
```
First Set: *Inline Function*

- **Inline Function**
  - A function's body is just as clear as its name.
  - *Put the function's body into the body of its callers and remove the function.*

```java
int getRating() {
    return (moreThanFiveLate Deliveries()) ? 2 : 1;
}

boolean moreThanFiveLateDeliveries() {
    return _numberOfLateDeliveries > 5;
}
```

```java
int getRating() {
    return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```
First Set: *Encapsulate Variable*

- **Encapsulate Variable**
  - You are accessing a variable directly, but the coupling to the variable is becoming awkward.
  - *Create getting and setting functions for the variable and use only those to access the variable.*

```java
private int _low, _high;
boolean includes (int arg) {
    return arg >= _low && arg <= _high;
}
```

```java
private int _low, _high;
boolean includes (int arg) {
    return arg >= getLow() && arg <= getHigh();
}
int getLow() {return _low;}
int getHigh() {return _high;}
```
First Set: *Introduce Parameter Object*

- **Introduce Parameter Object**
  - You have a group of parameters that naturally go together.
  - *Replace them with an object.*

```javascript
function amountInvoiced(startDate, endDate) {...}
function amountReceived(startDate, endDate) {...}
function amountOverdue(startDate, endDate) {...}

function amountInvoiced(aDateRange) {...}
function amountReceived(aDateRange) {...}
function amountOverdue(aDateRange) {...}
```
First Set: *Combine Functions into Class*

- **Combine Functions into Class**
  - A group of functions operate closely together on a common body of data
  - *Form a class to contain the functions;*

```javascript
function base(aReading) {...}
function taxableCharge(aReading) {...}
function calculateBaseCharge(aReading) {...}

class Reading {
    base() {...}
    taxableCharge() {...}
    calculateBaseCharge() {...}
}
```
First Set: *Split Phase*

- **Split Phase**
  - The code is dealing with two or more different things.
  - *Split it into separate modules.*

```javascript
const orderData = orderString.split(/\s+/);
const productPrice = priceList[orderData[0]].split("-")[1];
const orderPrice = parseInt(orderData[1]) * productPrice;

const orderRecord = parseOrder(order);
const orderPrice = price(orderRecord, priceList);

function parseOrder(aString) {
  const values = aString.split(/\s+/);
  return {
    productID: values[0].split("-")[1],
    quantity: parseInt(values[1]),
  };
}

function price(order, priceList) {
  return order.quantity * priceList[order.productID];
}
```
Encapsulation: Encapsulate Record

- Encapsulate Record
  - You have a mutable data record.
  - *Turn it into a class and create getting and setting methods to access the variables.*

```java
organization = {name: "Acme Gooseberries", country: "GB"};

class Organization {
    constructor(data) {
        this._name = data.name;
        this._country = data.country;
    }
    get name() {return this._name;}
    set name(arg) {this._name = arg;}
    get country() {return this._country;}
    set country(arg) {this._country = arg;}
}
```
Encapsulation: *Encapsulate Collection*

- **Encapsulate Collection**
  - A method returns a collection.
  - *Make it return a read-only view and provide add/remove methods.*
Encapsulation: *Replace Primitive with Object*

- **Replace Primitive with Object**
  - You have a data item that needs additional data or behavior.
  - *Turn the data item into an object.*
Encapsulation: *Extract Class*

- **Extract Class**
  - You have one class doing work that should be done by two.
  - *Create a new class and move the relevant fields and methods from the old class into the new class.*
Encapsulation: *Inline Class*

**Inline Class**

- A class isn't doing very much.
- *Move all its features into another class and delete it.*
Encapsulation: *Hide Delegate*

- **Hide Delegate**
  - A client is calling a delegate class of an object.
  - *Create methods on the server to hide the delegate.*
Encapsulation: *Remove Middle Man*

- **Remove Middle Man**
  - A class is doing too much simple delegation.
  - *Get the client to call the delegate directly.*

![Diagram showing encapsulation](image)
Reference