Refactoring & Anti-Patterns

CSC207 Software Design

Refactoring

Changing the code **without** changing its functionality.

Refactor to fix problems with the design.

Programmers often refactor their code to make it easier to then implement a new feature.

A few refactoring techniques

- Extract Method
- Change Method Declaration
- Encapsulate Fields
- Split Loop
- Slide Statements
- Replace Constructor with Builder
- Replace Constructor with Factory Method

A few refactoring techniques

- Extract Method
- Change Method Declaration
- Encapsulate Fields
- Split Loop
- Slide Statements
- Replace Constructor with Builder
- Replace Constructor with Factory Method

Performing a sequence of refactoring steps can lead to better code — while each step makes just a small change so that it is easy to avoid introducing bugs.

Advice: test after each change to the code!

Extract Method

Steps:

- Create a helper method containing the code for one step of a longer method.
- Update the original method to call the helper.

Benefits:

- Improves readability

Change Method Declaration

Steps:

- Modify the parameter list to a method.
- Update the method body as needed.
- Update any calls to the method.

Benefits:

- Adds flexibility (by adding parameters) or hide details (by removing parameters)

Encapsulate Fields

Steps:

- Change instance variable visibility to private.
- Add getters or setters as appropriate.
- Update client code to use setters and getters.

Benefits:

- Controls access to instance variables.

Split Loop

Steps:

- Move independent operations from one loop into separate loops.

Benefits:

- Improves readability and can make other refactoring techniques, like extract method, easier to perform.

Slide Statements

Steps:

- Reorder independent operations.

Benefits:

- Improves readability and can make other refactoring techniques, like extract method, easier to perform.
- Group together related operations.

Replace Constructor with Builder

Steps:

- Create a Builder class with setters to customize the object being built and a method to build the object.
- Update constructor calls in the client code to use the Builder instead.

Benefits:

- Improves readability of code.
- Can replace the need for many overloaded constructors for customizable classes.

Replace Constructor with Factory Method

Steps:

- Define a new method that returns an instance of the class.
- Hide the constructor from the client and have them use the factory method instead.

Benefits:

- Hides explicit calls to the constructor

Anti-Patterns

- Anti-patterns refer to code that would benefit from having a design pattern applied to it.

- Through a sequence of refactoring steps, we can transform code that follows an anti-pattern into code that follows an appropriate design pattern.

- The course notes contains examples of some anti-patterns for the design patterns covered in this course.

Example

```
public class Student {
    public int studentNumber;
}
```

```
public class Student {
    private int studentNumber;

    public Student(int number) {
        this.studentNumber = number;
    }

    public int getStudentNumber() {
         return studentNumber;
    }
}
```

Example

```
public class Pizza {
    private final String size;
    private final boolean cheese;
    private final boolean pepperoni;
    public Pizza(String size) { this(size, false); }
    public Pizza(String size, boolean cheese) {  this(size, cheese, false); }
    public Pizza(String size, boolean cheese, boolean pepperoni) {
        this.size = size;
        this.cheese = cheese;
        this.pepperoni = pepperoni;
    }

    public static void main(String[] args) {
        Pizza pizza = new Pizza("Large", true, true);
    }
}
```

```
public class PizzaBuilder {
    private String size;
    private boolean cheese = false;
    private boolean pepperoni = false;

    public PizzaBuilder setSize(String size) {
        this.size = size;
        return this;
    }
    public PizzaBuilder addCheese() {
        this.cheese = true;
        return this;
    }
    public PizzaBuilder addPepperoni() {
        this.pepperoni = true;
        return this;
    }
    public Pizza build() {
        return new Pizza(size, cheese,
                         pepperoni);
    }
}
```

```
public class Pizza {
    private final String size;
    private final boolean cheese;
    private final boolean pepperoni;

    public Pizza(String size,
                 boolean cheese, boolean pepperoni) {
        this.size = size;
        this.cheese = cheese;
        this.pepperoni = pepperoni;
    }

    public static void main(String[] args) {
        Pizza pizza = new PizzaBuilder()
                          .setSize("Large")
                          .addCheese()
                          .addPepperoni()
                          .build();
    }
}
```

Example

```
public class RefactoringExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
        List<String> upperNames = new ArrayList<>();
        int totalLength = 0;

        for (String name : names) {
            upperNames.add(name.toUpperCase());
            totalLength += name.length();
        }

        System.out.println("Uppercase Names: " + upperNames);
        System.out.println("Total Length: " + totalLength);
    }
}
```

```
public class RefactoringExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
        List<String> upperNames = new ArrayList<>();
        int totalLength = 0;

        for (String name : names) {
            upperNames.add(name.toUpperCase());
        }
        for (String name : names) {
            totalLength += name.length();
        }

        System.out.println("Uppercase Names: " + upperNames);
        System.out.println("Total Length: " + totalLength);
    }
}
```

```
public class RefactoringExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

        List<String> upperNames = new ArrayList<>();
        for (String name : names) {
            upperNames.add(name.toUpperCase());
        }
        System.out.println("Uppercase Names: " + upperNames);

        int totalLength = 0;
        for (String name : names) {
            totalLength += name.length();
        }
        System.out.println("Total Length: " + totalLength);
    }
}
```

```
public class RefactoringExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

        printUppercaseNames(names);
        printTotalLength(names);
    }
    private static void printUppercaseNames(List<String> names) {
        …
        System.out.println("Uppercase Names: " + upperNames);
    }
    private static void printTotalLength(List<String> names) {
        …
        System.out.println("Total Length: " + totalLength);
    }
}
```