2 March 2023

Refactoring Report of ChemistryCalculator

Submitted To
Dipok Chandra Das
Assisstant Professor, IIT, NSTU

Team Members -
Emran Hossain [ASH1825002M]
Md Azad Hossain [ASH1825014M]
Al Shahriar Priyo [ASH1825020M]
Nadim Bhuiya [ASH1825034M]
Akash Chandra Debnath[ASH1825037M]

1. Refactoring

Refactoring means performing changes to the structure of software to make it easier to comprehend and cheaper to make subsequent changes without changing the observable behavior of the system. Restructuring is a related concept for non-object-oriented systems. Refactoring is achieved through removal of duplicate code, simplification of code, and moving code to a different class, among others. With- out continual refactoring, the internal structure of software will eventually deform beyond comprehension, due to periodic maintenance.

1.1 Why Refactoring is Important?

- To improve the design of applications.
- To make software easier to understand.
- To find bugs
- To make the program run faster.
- To fix existing legacy database
- To support revolutionary development
- To provide greater consistency for user

2. Benefits of Refactoring

- ➢ Makes code more readable.
- ➢ Cleanup code and make it tidier.
- ➢ Removes redundant, unused code and comments.
- ➢ Improves performance.
- ➢ Makes some things more generic.
- ➢ Keeps code DRY (Don’t Repeat Yourself)
- ➢ Combine and dispose of “Like” or “Similar” code.
- ➢ Splitting out long functions into more manageable bites.
- ➢ Create reusable code.
- ➢ Better class and function cohesion.

3. Activities in a Refactoring Process

To restructure a software system, programmers follow a process with well defined activities. Those activities are as follows:

A. Identify what to refactor.
B. Determine which refactorings should be applied.
C. Ensure that refactoring preserves the software's behavior.
D. Apply the refactorings to the chosen entities.
E. Evaluate the impacts of the refactorings.
F. Maintain consistency.

A. Identify What to Refactor

The programmer chooses what to refactor from a collection of software artifacts in this step.The programmer may take into account software artifacts like source code, design documentation, and requirements documents. After determining the top level item, the programmer can concentrate on refactoring certain sections of the selected artifact. The source code can be used to locate particular modules, functions, classes, methods, and data structures that can be refactored.

B. Determine Which Refactorings Should be Applied

The programmer decides which refactorings to apply to the software components that were identified in the first phase outlined above in this stage.

C. Ensure that Refactoring Preserves the Behavior of the Software

The behavior of a program should, in theory, be the same after refactoring as it was before.Program behavior was merely input-output behavior in Opdyke's initial concept of behavior preservation [1].In other words, it was wanted for the programs before and after restructuring to produce the same output values for the same set of input values.

D. Apply the Refactorings to the Chosen Entities

This entails carrying out the stages of the prior refactorings that were selected.

E. Evaluate the Impacts of the Refactorings on Quality

Refactorings have an effect on both internal and exterior qualities.Size, complexity, coupling, coherence, and testability are some instances of internal attributes. Performance, reusability, maintainability, extensibility, robustness, and scalability are a few examples of external attributes.

F. Maintain Consistency

Many artifacts at various degrees of abstraction can all be used to describe a software system.These artifacts include of test suites, source code, requirements documents, and design documents.In order to preserve consistency among the artifacts, it is crucial to alter some or all of the others whenever one type of item is changed.For instance, modifications to the source code may necessitate alterations to the design specifications and test plans.

4. Two techniques to Analyze Refactoring

1) Critical pair analysis
2) Sequential dependency analysis

1) Critical Pair Analysis

Here, the goal is to find pairs of refactorings that are mutually exclusive.Check each pair of refactorings for conflicts given a set of refactorings.If two refactorings cannot be applied simultaneously, they are said to be conflicting.

2) Sequential Dependency Analysis

Sequential dependency of refactorings indicates that:

- Before applying a refactoring, one or more refactorings must have been applied, and
- After applying a refactoring, a refactoring that is mutually exclusive cannot be applied.

5. Code Smell

A code smell is any symptom in the source code of a software that possibly indicates a deeper problem.Code smells are something that the majority of application developers and testers eventually run into, especially when working with complicated applications or in large teams.These are apparent and palpable signs that there is a problem with an application's core code that can potentially result in catastrophic failures and ruin the performance of an application.

Some examples of code smells:

- Duplicate Code
- Long Parameter List
- Long Methods
- Large Classes
- Message Chain

Duplicate Code

This odour appears when sections of source code are repeated throughout the program.If the duplicate code needs to be replaced, it must be made sure that all segments are changed.Faults will be introduced if a section is missing, which is likely to happen.The nature of code duplications holds the answers.For instance, duplicates can be extracted into a separate method if they appear in many methods of the same class.

Long Parameter List

When a method contains more formal parameters than it needs, let's say four or more, programmers may make mistakes when creating method calls.Rearranging the list of actual parameters is a frequent mistake.Designing a parameter object, which allows a single parameter to be passed instead of a large list of parameters, might be the answer.

Long Methods

This happens if a method contains a lengthy string of statements, such as several hundred lines of code.Extraction of methods from lengthy code snippets offers a solution.

Large Classes

If a class has more than eight methods and more than fifteen variables, for example, there is considered to be a stench.The class may be solved by breaking it up into component classes and adding superclasses.

Message Chain

When one successfully calls many methods, a message chain results. A communication chain looks like this: student.getID().getRecord().getGrade (course)**.** By using a helper function that completes a portion of the calculations of another function, such a chain can be made simpler.Thus, student.getGrade can be used to replace the preceding message chain (course).

Refactoring Techniques To Improve Software

Refactoring is a multidimensional process that we could perform through either of there have discussed four techniques:

- Extract Method
- Simplifying Methods
- Composing Method
- Abstraction

Extract Method

The main goal of the Extract Method is to make your code more readable overall by resolving complexity, offering clarifications, and regularizing the structure.The actual process comprises moving a piece of code from its established method into a new one.In order to make it simple for development teams to put together the entire source code structure, the second set of methods are named properly to clarify the functions of the code blocks.

Simplifying Methods

Simplifying Methods are designed to make code simpler, whereas the Extract Method improves readability.This is yet another refactoring technique that allows multiple approaches, as the phrase "same" suggests. The two choices you have are as follows:

- Simplifying Method Calls Refactoring
- Simplifying Conditional Expressions Refactoring

Composing Method

When you encounter code routines that are a little too lengthy and challenging to understand or execute, you turn to the Composing Method.The method is designed to rearrange the code by removing duplicates or, alternatively, to split it up into portable chunks.

Abstraction

Abstraction, a restructuring method that is best suited for massive projects, comes in at number five on our list. Abstraction basically aims to eliminate redundant and repetitive code from your program.It does it by employing strategies like extraction, interface creation, class creation, hierarchy, class inheritance, etc.

As refactoring might need to **change in source code** to get better performance and eliminate code smells, we needed to take care of **change impact analysis**. Without the CIA it might dive to a worser state and might be unable of the **ChemistryCalculator** application.

7. Change Impact Analysis (CIA)

Change Impact Analysis is the process of investigating the undesired consequences of a change in a software module and is considered of paramount importance, in the sense that it aims to reduce the risk of software failure and the maintenance cost.

8. Impact Analysis Process

Change impact analysis is the process of identifying the potential impact of a change on different aspects of a system, including the requirements, design, code, test cases, and other related artifacts. It involves analyzing the relationships and dependencies between different components of a system to determine the potential impact of a change.

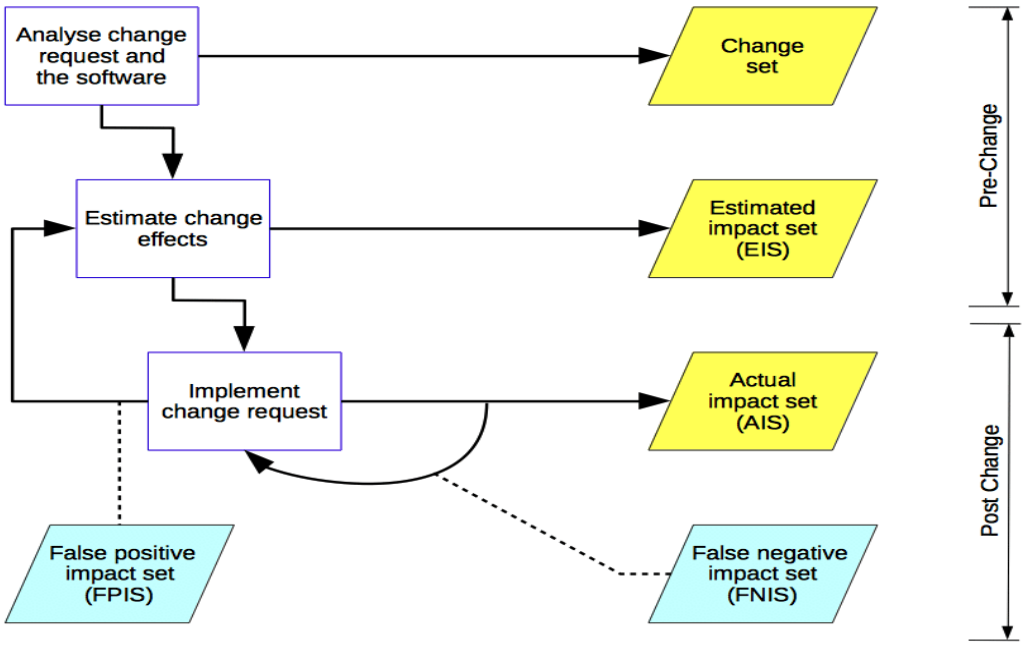


Fig:Impact Analysis Process

The various sets of components are formally defined as follows:

8.1 Starting Impact Set (SIS): The initial set of objects (or components) presumed to be impacted by a software CR is called SIS.

8.2 Candidate Impact Set (CIS): The set of objects (or components) estimated to be impacted according to a certain impact analysis approach is called CIS. Software lifecycle objects (SLOs), which are added to the SIS, are subject to change as SIS constituent parts do.A software system's various components may be directly or indirectly impacted by changes to one of them. The following provides an explanation of both direct impact and indirect influence:

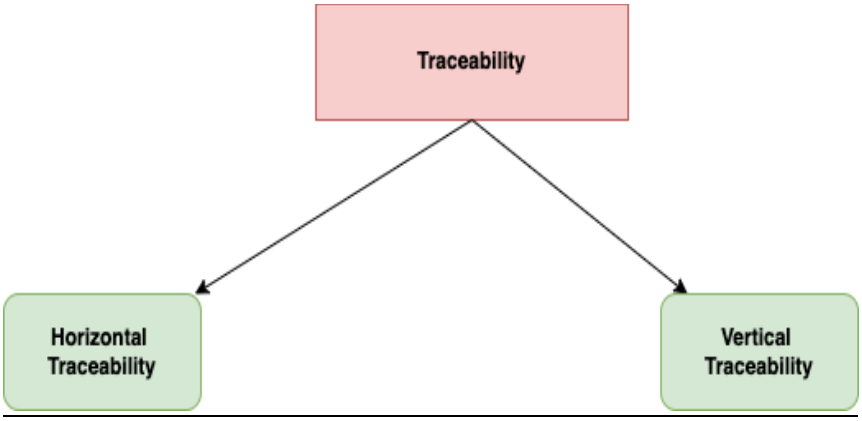
1) **Direct impact:** A direct impact relation exists between two entities, if the two entities are related by a fan-in and/or fan-out relation.
2) **Indirect impact:** The term "indirect impact" describes changes that occur later as a result of a direct impact, such as losing housing if a structure was uninhabitable. We can state that an entity A indirectly affects C if entity B directly affects entity C and entity C directly affects entity A.

8.3 Discovered Impact Set (DIS): DIS is defined as the set of new objects (or components), not contained in CIS, discovered to be impacted while implementing a CR.

8.4 Actual Impact Set (AIS): The set of objects (or components) actually changed as a result of performing a CR is denoted by AIS.

8.5 False Positive Impact Set (FPIS): FPIS is defined as the set of objects (or components) estimated to be impacted by an implementation of a CR but not actually impacted by the CR. Precisely, FPIS = (CIS ∪ DIS) \ AIS.

9. Traceability Management for Impact Analysis



9.1 Vertical Traceability

The capacity to track dependent artifacts within a model is referred to as vertical traceability. It's also known as internal traceability.

9.2 Horizontal Traceability

The term "horizontal traceability" describes the capacity to track objects across many models. It's also known as external traceability.

10. Dependency-Based Impact Analysis

Source code objects are typically examined to gather vertical traceability data.Syntactic dependencies are likely to lead to semantic dependencies, hence dependency-based impact analysis tools examine syntactic dependencies to determine the impact of changes. Two traditional impact analysis techniques -

1. Call Graph
2. Program Dependency Graph

10.1 Call Graph

In a call graph, each node stands for a component, method, or function, and any edges between nodes A and B denote the possibility of A invoking B. Call graphs are used by programmers to comprehend the potential effects that a software modification might have.

10.2 Program Dependency Graph

A program is represented by the PDG (Program Dependency Graph) as a graph, where the nodes are statements and predicate expressions (or operators and operands), and the edges incident to a node are both the data values that the node's operations depend on and the control conditions that determine whether the operations are carried out. It can be further classified into two program slicing techniques

I. Static Program Slicing
II. Dynamic Program Slicing

I. A **static program slice** is identified from a PDG as follows: (i) for a variable var at node n, identify all reaching definitions of var; and (ii) find all nodes in the PDG which are reachable from those nodes. The visited nodes in the traversal process constitute the desired slice.
II. A simple way to find **dynamic slices** is as follows: (i) for the current test, mark the executed nodes in the PDG; and (ii) traverse the marked nodes in the graph.

We followed the **static program slicing technique** in our refactoring work.

Refactoring Codes for ChemistryCalculator Frontend Classes

ClassName: TitrationPanel.java

1. Code smell: Long Method (Too many lines of code inside one method)

Solution: Extract Method

```
private GroupLayout extracted() {
        GroupLayout notificationPanelLayout = new GroupLayout(notificationPanel);
        notificationPanel.setLayout(notificationPanelLayout);
        notificationPanelLayout.setHorizontalGroup(
            notificationPanelLayout.createParallelGroup(GroupLayout.Alignment.LEADING)
                .addGroup(GroupLayout.Alignment.TRAILING,
notificationPanelLayout.createSequentialGroup()
                    .addGap(0, 0, Short.MAX_VALUE)
                    .addComponent(notificationLabel, GroupLayout.PREFERRED_SIZE, 654,
GroupLayout.PREFERRED_SIZE))
        );
        return notificationPanelLayout;
    }
```

ClassName: PercentofCompletionPanel.java

2. Code Smell: Duplicate Code

Solution: Push Up Method

```
private void extracted5() {
        ansTable.setModel(dataTableModel);
        ansTable.setCursor(new Cursor(Cursor.TEXT_CURSOR));
        ansTable.setRowHeight(50);
        ansTable.setShowVerticalLines(false);
}

private void extracted4() {
        clearButton.setBackground(MAIN_COLOR);
        clearButton.setFont(SEGOE_UI);
        clearButton.setForeground(GRAY);
        clearButton.setText("Clear");
        clearButton.setAutoscrolls(true);
        clearButton.setCursor(new Cursor(Cursor.HAND_CURSOR));
        clearButton.addActionListener(this::pcom_clear_buttonActionPerformed);
}
```

3. Code smell: Divergent Change
Solution: Extract class

```
public void setDataTableModel(DefaultTableModel dataTableModel) {
        this.dataTableModel = dataTableModel;
}
```

4. Code Smell: Long Parameter List
Solution: Nested Class

```
dataTableModel = new DefaultTableModelExtension(new Object[][]{}, new String[]{});
```

5. Code Smell: Long Method
Solution: Introduce Parameterized Method

```
 GroupLayout Layout = new GroupLayout(this);
        this.setLayout(Layout);
        extracted6(Layout);

    }
private void extracted6(GroupLayout Layout) {
        Layout.setHorizontalGroup
```

Classname: ElectronConfigPanel.java

6. Code Smell: Conditional Statements
Solution: Decompose Conditional

```
if (!atom_text.isEmpty()) {
            Atom atom;
            try {
                final int atomicNumber = Integer.parseInt(atom_text);
                try {
                    atom = Atom.getInstance(atomicNumber);
                    extracted(atom);
```

7. Codesmell: Conditional Statements
Solution: Decompose Conditional

```
catch (NumberFormatException e) {
            try {
                atom = Atom.getInstance(atom_text);
                extracted2(atom);

                errorMessagePanel.setVisible(false);
```

8. Code Smell: Duplicate Code
Solution: Extract Method

```
GroupLayout Layout = new GroupLayout(this);
        this.setLayout(Layout);
        extracted3(Layout);
        extracted4(Layout);
```

Classname: EquationBalancePanel.java

9. Code Smell: Long Method
Solution: Extract Method

```
private void initComponent() {
        extracted();
        extracted2();
        extracted3();
        extracted4();
private void extracted2() {
        errorMessagePanel.setBackground(Color.red);
```

```
        errorMessagePanel.setVisible(false);

        errorMessageLabel.setFont(SEGOE_UI);
        errorMessageLabel.setForeground(Color.white);
        errorMessageLabel.setHorizontalAlignment(SwingConstants.CENTER);
}

private void extracted() {
        labelForReactantsTextfield.setFont(SEGOE_UI);
        labelForReactantsTextfield.setForeground(MAIN_COLOR);
        labelForReactantsTextfield.setText("Reactants : ");


        labelForProductsTextfield.setFont(SEGOE_UI);
        labelForProductsTextfield.setForeground(MAIN_COLOR);
        labelForProductsTextfield.setText("Products  :");
}
```

10. CodeSmell: Duplicate Code
Solution: Created Interface to reduce code.

```
package ChemistryCalculator.frontend;
import java.lang.foreign.GroupLayout;
public interface setComponentLayout {
    GroupLayout Layout = new GroupLayout(this);
    @Override
    boolean equals(Object obj)
        this.setLayout(Layout);
```

Classname: MolarMassPanel.java

11. Code Smell:Long Parameter List
Solution: Nested Class

```
dataTableModel = new DefaultTableModelExtension(new Object[][]{}, new String[]{});
```

ClassName: Formater.java

12. Code Smell: Long Class

Solution: Decompose Conditionals

```
  while (!matcher.find()) {
        String group = matcher.group();
        if (group.length() == 1) {
           int number = Integer.parseInt(group);
           output.append(segments[i]).append(subScript[number]);
        } else {

           char[] number = group.toCharArray();
           String result = IntStream.range(0, number.length).mapToObj(j ->
subScript[Character.getNumericValue(number[j])]).collect(Collectors.joining());

           output.append(segments[i]).append(result);
        }
```

Classname: PercentofCompletion.java

13. Code Smell: Primitive Obsession

Solution: Replace Data Value with Object

```
public JPanel getErrorMessagePanel() {
      return errorMessagePanel;
}

public JLabel getErrorMessageLabel() {
      return errorMessageLabel;
}
```

```
private DefaultTableModel dataTableModel;

   public DefaultTableModel getDataTableModel() {
       return dataTableModel;
}

public void setDataTableModel(DefaultTableModel dataTableModel) {
       this.dataTableModel = dataTableModel;
}
```

14. Code Smell: Long Method
Solution: Extract Method

```
private void extracted5() {
       ansTable.setModel(dataTableModel);
       ansTable.setCursor(new Cursor(Cursor.TEXT_CURSOR));
       ansTable.setRowHeight(50);
       ansTable.setShowVerticalLines(false);
}
```

```
  private void initComponent() {
       extracted();
       extracted2();
```

Classname: Home.java

15. Code Smell: Long Method
Solution: Extract Method

```
private void buildBodyPanels() {
       //adding all body panels to a single panel (bodyPanel)
       extracted2();
```

```
  }
```

16. Code Smell: Duplicate Code

Solution: Push up Method

```
 private void buildSidebar() {
      // creating sidebar. Every single body panel should have a single menu bar for
navigation. Don't creates duplicates.

      extracted();

      sidebarPanel.build();
  }
```

Refactoring Codes for ChemistryCalculator Backend Classes

Refactored Class - Atom.java

1. Code Smell : Duplicated Code.

Extract Method: The **getInstance** methods perform similar tasks. The common code can be separated into a distinct private method and called from both getInstance methods.

```
private createAtom(String symbol, int atomicNumber) {
    if (symbol != null && isValid(symbol)) {
        return new Atom(symbol);
    } else if (atomicNumber >= 1 && atomicNumber <= 118) {
        return new Atom(atomicNumber);
    }
    throw new InvalidAtomException("Invalid symbol or atomic number");
}

public static Atom getInstance(String symbol) {
    return createAtom(symbol, 0);
}

public static Atom getInstance(int atomicNumber) {
    return createAtom(null, atomicNumber);
}
```

2. **Code Smell**: Primitive Obsession (using primitives for simple tasks rather than small objects).

Use forEach: The entries are already sorted by the atomic number, so the **forEachOrdered** method in the **Atom constructor**, which requires an **atomicNumber** parameter, is not required. Instead, we can use the more effective **forEach** method.

```
private Atom(int atomicNumber) {
    allAtoms.entrySet().stream()
            .filter(entry -> atomicNumber == Integer.parseInt(entry.getValue()[0]))
            .forEach(entry -> {
                this.symbol = entry.getKey();
                this.name = entry.getValue()[1];
                this.atomicMass = Double.parseDouble(entry.getValue()[2]);
            });
    this.atomicNumber = atomicNumber;
  }
```

3. **Code Smell** : Long method with generalization problem.

Calculate electron configuration more simply: By using a loop and an array to represent the orbitals, the getElectronConfig method can be made simpler.

```
public String getElectronConfig() {
    int[] orbitalSizes = {2, 2, 6, 2, 6, 2, 10, 6, 2, 10, 6, 2, 14, 10, 6, 2, 14, 10, 6, 2};
    StringBuilder output = new StringBuilder();
    int remainingElectrons = atomicNumber;
    for (int i = 0; i < orbitalSizes.length; i++) {
```

```
        int electronsInOrbital = Math.min(orbitalSizes[i], remainingElectrons);
        output.append((i + 1)).append("s").append(electronsInOrbital).append(" ");
        remainingElectrons -= electronsInOrbital;
        if (remainingElectrons == 0) {
            break;
        }
    }
    return output.toString();
}
```

Refactored Class - Compound.java

4. Since the implementation of the list isn't crucial and may change in the future, the declaration of **percentageOfCompletion** was changed from **ArrayList** to **List**.

```
private List<Map<String, String>> percentageOfCompletion;
```

5. Changed the names of the variables **atomicMass** and **v** to **coefficient** and **mass**, respectively, in order to give them names that were more descriptive.

```
private void calculateMolarMass() {
    double sum = 0;
    for (Map.Entry<Atom, Integer> entry : compoundManager.getAtomList().entrySet()) {
        Atom atom = entry.getKey();
        int coefficient = entry.getValue();
        int totalAtoms = compoundManager.getElementMatrix().get(0).get(coefficient);
        double atomicMass = totalAtoms * atom.getAtomicMass();
        sum += atomicMass;
    }
    molarMass = sum;
```

6. Removed the unnecessary '**this**' keywords from the method calls for **calculateMolarMass()** and **getMolarMass** ().

```
public double getMolarMass() {
    if (molarMass == null) {
        calculateMolarMass();
    }
    return molarMass;
}
```

7. Combined the logic in **calculatePercentageOfCompletion()** by calculating the **massPercent** and creating the **elementDetails** map in the same loop.

```
private void calculatePercentageOfCompletion() {
    percentageOfCompletion = new ArrayList<>();
    for (Map.Entry<Atom, Integer> entry : compoundManager.getAtomList().entrySet()) {
        Atom atom = entry.getKey();
        int coefficient = entry.getValue();
        int totalAtoms = compoundManager.getElementMatrix().get(0).get(coefficient);
        double atomicMass = totalAtoms * atom.getAtomicMass();
        double massPercent = atomicMass / getMolarMass() * 100;

        Map<String, String> elementDetails = new HashMap<>();
        elementDetails.put("symbol", atom.getSymbol());
        elementDetails.put("name", atom.getName());
        elementDetails.put("atomicMass", String.valueOf(atomicMass));
        elementDetails.put("totalAtoms", String.valueOf(totalAtoms));
        elementDetails.put("massPercent", String.valueOf(massPercent));

        percentageOfCompletion.add(elementDetails);
    }
}
```

Refactored Class - CompoundManager.java

8. **Code Smell**: Unnecessary Commented Code

Delete Commented Code: Erase these commented Code for making a neat and clean code format and also optimize memory storage.

Refactored Class - Concentration.java

9. Code Smell: Code repetition and ambiguity.

Moved the calculation of **compound.getMolarMass()** to a local variable **molarMass** to improve readability and eliminate code repetition.

```
public double getMolarity() {
    double molarMass = compound.getMolarMass();
    return (1000 * givenCompoundMass) / (molarMass * volumeOfSolution);
}
```

10. **Code Smell**: Coupling Methods

Changed the **geNormality()** method to calculate normality using the **getMolarity()** method.

```
public double geNormality(double equivalentNumber) {
    double molarity = getMolarity();
    return molarity * equivalentNumber;
}
```

11. **Code Smell**: Unnecessary Commented Code

Delete Commented Code: Erase these commented Code for making a neat and clean code format and also optimize memory storage.

Refactored Class - Converter.java

12. Changed the **registerFactor** and **putFactor** methods to use a single **putFactor** method that takes both **from** and **to** units.

```
private static void putFactor(String from, String to, double factor) {
    FACTOR_MAP.computeIfAbsent(from, k -> new HashMap<>()).put(to, factor);
  }
```

13. Simplified the **convert** method by finding the conversion factors for **from** and **to** units using **Optional** and then computing the result.

```
public static double convert(String from, String to, double value) {
    Optional<Map.Entry<String, Double>> first =
Optional.ofNullable(FACTOR_MAP.get(from))
            .orElseThrow(() -> new IllegalArgumentException("Cannot convert from " +
from));
```

```
    Optional<Map.Entry<String, Double>> second =
Optional.ofNullable(FACTOR_MAP.get(to))
            .orElseThrow(() -> new IllegalArgumentException("Cannot convert to " + to));
    double firstFactor = first.filter(entry -> entry.getKey().equals(to))
            .map(Map.Entry::getValue)
            .orElseThrow(() -> new IllegalArgumentException("Cannot convert from " + from
+ " to " + to));
    double secondFactor = second.filter(entry -> entry.getKey().equals(from))
            .map(Map.Entry::getValue)
            .orElseThrow(() -> new IllegalArgumentException("Cannot convert from " + from
+ " to " + to));
    return value * firstFactor / secondFactor;
  }
```

Refactored Class - EquationBalancer.java

14. Split the string inputs into arrays and trimmed the array elements in one step using **Arrays.stream()** method and **map()** method.

```
String[]reactants
=Arrays.stream(this.reactants.replaceAll("\\s+","").split("\\+")).map(String::trim).toArray(Stri
ng[]::new);

String[]products =
Arrays.stream(this.products.replaceAll("\\s+","").split("\\+")).map(String::trim).toArray(Strin
g[]::new);
```

15. Combined the two **for loops** into a single loop and reduced class size

```
    StringBuilder result = new StringBuilder();
    for (int i = 0; i < reactants.length; i++) {
        if (i > 0) {
            result.append(" + ");
        }
        result.append(balancedCoefficients[i]);
        result.append(reactants[i]);
    }
```

16. **Code Smell**: Inappropriate Variable Name

Changed the name of the balanced coefficients array from **balancedCoefficient** to **balancedCoefficients**.

Refactored Class - Fraction.java

17. **Dividing by Zero** is caused mathematically **error** so we have to check that the divider is not zero and if it is zero then it is called an **exception**.

```
public Fraction divide(Fraction fractionTwo) {
    if (other.numerator == 0) {
        throw new IllegalArgumentException("Cannot divide by zero.");
    }
    int newNumerator = numerator * fractionTwo.getDenominator();
    int newDenominator = denominator * fractionTwo.numerator;
    return new Fraction(newNumerator, newDenominator);
}
```

18. Checked exception and more robustly structured **reduce()** method

```
private void reduce() {
    if (denominator == 0) {
        throw new IllegalStateException("Denominator cannot be zero.");
    }
    int gcd = getGCD(numerator, denominator);
    numerator /= gcd;
    denominator /= gcd;
    if (denominator < 0) {
        numerator *= -1;
        denominator *= -1;
    }
}
```

*It can be declared as public so that other classes can access it.

19. Removed unnecessary conditions from the **getGCD**() method and now it will work more smoothly.

```
private static int getGCD(int a, int b) {
    a = Math.abs(a);
    b = Math.abs(b);
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
```

20. Denominator **can't be zero** so we need to **throw an exception** here for the zero value unless it should fail to calculate the result.

```
public void setDenominator(int denominator) {
    if (denominator == 0) {
        throw new IllegalArgumentException("Denominator cannot be zero.");
    }
    this.denominator = denominator;
    reduce();
}
```

Refactored Class - Titration.java

21. Changed the String arrays to **List<Double>** to simplify the code

```
private final List<Double> acidProperties = new ArrayList<>();
private final List<Double> baseProperties = new ArrayList<>();
```

22. Added a **constructor** to add all the properties at once

```
public Titration(Double molarityOfAcid, Double molarityOfBase, Double volumeOfAcid,
                 Double volumeOfBase, Double numOfMolesOfAcid, Double numOfMolesOfBase) {
    addAcidProperties(molarityOfAcid, volumeOfAcid, numOfMolesOfAcid);
    addBaseProperties(molarityOfBase, volumeOfBase, numOfMolesOfBase);
}
```

N.B. Unfortunately it produced another code smell - Long Parameter List.

23. Removed the **unknownValue_in_Acid flag** and replaced it with a **null check** to determine the unknown value index

```
    if (isValid()) {
        double numerator = 1;
        double denominator = 1;
        int unknownIndex = -1;
        if (acidProperties.contains(null)) {
            unknownIndex = acidProperties.indexOf(null);
            for (Double baseProperty : baseProperties) {
                numerator *= baseProperty;
            }
```

24. Simplified the **isValid()** method by using a **null** count instead of checking for e**mpty strings**.

```
// Check if the input is valid
private boolean isValid() {
    int nullCount = 0;
    for (Double property : acidProperties) {
        if (property == null) {
            nullCount++;
        }
    }
    for (Double property : baseProperties) {
        if (property == null) {
            nullCount++;
        }
    }
    return nullCount == 1;
}
```