Investigating the Effectiveness of Greedy Algorithm on Open Source Software Systems for Determining Refactoring Sequence

Sandhya Tarwani\textsuperscript{1} \hspace{1cm} Ashish Sureka\textsuperscript{2}

\textsuperscript{1}SRM University, India (sandhya.tarwani@gmail.com)

\textsuperscript{2}Ashoka University, India (ashish.sureka@ashoka.edu.in)

QuASoQ 2017 (co-located to APSEC 2017)
Table of Contents

1 Research Motivation and Aim
   • Background and Context Setting
   • Research Contributions

2 Related Work

3 Solution Approach
   • N-Ary Trees and Greedy algorithm
   • One-ary analysis
   • Two-ary analysis

4 Dataset and Results
   • Experimental Dataset
   • Experimental Results

5 Conclusion
Table of Contents

1 Research Motivation and Aim
   • Background and Context Setting
     • Research Contributions

2 Related Work

3 Solution Approach
   • N-Ary Trees and Greedy algorithm
   • One-ary analysis
   • Two-ary analysis

4 Dataset and Results
   • Experimental Dataset
   • Experimental Results

5 Conclusion
**Code Smells and Refactoring**

**Code smells** are indicators of root problem in the source code [2].

**Refactoring** is a term used for the restructuring and redesigning of the existing code without altering its external attributes.

Fowler [2] defined more than 70 types of refactoring techniques like extract method, extract class etc.

Refactoring helps in transforming the source code that no longer contains code smell.
Several researchers are conducting study on finding the **correct or best sequence for refactoring techniques** so that software maintainability value gets enhanced [6][7][13].

If the sequence is known in advance to the software developers, then it will substantially reduce the effort and time spent on bug fixing and thereby improving quality of the software.
Table of Contents

1. Research Motivation and Aim
   - Background and Context Setting
   - Research Contributions

2. Related Work

3. Solution Approach
   - N-Ary Trees and Greedy algorithm
   - One-ary analysis
   - Two-ary analysis

4. Dataset and Results
   - Experimental Dataset
   - Experimental Results

5. Conclusion
Novel and Unique Contributions

Novel Algorithm
A greedy-approach based algorithm for determining the refactoring sequence for a software system. Our study is the first work on grouping classes based on the number of bad smells and then applying the greedy algorithm.

Empirical Validation
An empirical analysis on four open-source software systems to exhibit the effect of the proposed approach. Our study is the first work on JTDS, JChess, OrDrumbox and ArtOfIllusion dataset.
A. Ghannem et al. [3] proposed an approach for automating the refactoring process in the source code with the help of Iterative Genetic Algorithm.

Y. Khrishe and M. Alshayeb [5] conducted an empirical study to find out whether order of applying refactoring affects the quality of the software or not.
I. Toyoshima et al. [11]
I. Toyoshima et al. [11] proposed 3 gate refactoring algorithm which is a new refactoring algorithm that is developed with the help of three refactoring rules of Workflow net.

A. Shahjahan et al. [8]
A. Shahjahan et al. [8] used graph theory techniques to propose a new method of code refactoring which is applied on projects written in Java language.
G. Szoke et al. [9]

G. Szoke et al. [9] developed a refactoring toolset called FaultBuster that helps in detecting problems in source code with the help of source code analysis, running automatic algorithms to remove bad smells and execute integrated testing tools.

Meananeatra et al. [6]

Meananeatra et al. [6], Eduardo et al. [7] and Wongpiang et al. [13], Tarwani et al. [10] present techniques on searching for refactoring sequence for single class of a dataset.
Table of Contents

1. Research Motivation and Aim
   - Background and Context Setting
   - Research Contributions

2. Related Work

3. Solution Approach
   - N-Ary Trees and Greedy algorithm
     - One-ary analysis
     - Two-ary analysis

4. Dataset and Results
   - Experimental Dataset
   - Experimental Results

5. Conclusion
We download source-code dataset from sourceforge and bad smells are identified with the help of plug-ins like JDeodorant [9][12]. Classes are then prioritized on the basis of number of bad smells. Only those classes are considered whose number of bad smells are greater than or equal to 4.

1-ary tree is formed where number of refactoring techniques is less than or equal to 3 and 2-ary tree is formed otherwise. After the formation of the trees, greedy algorithm is used to find out the best sequence for maximizing maintainability.
Proposed Solution Approach - Multi-Step Process

1. Identify bad smell
2. Categorize the classes and pick one class from each category
3. Calculate sum of metric values
4. Observe change in metric value
5. Apply refactoring techniques
6. Calculate sum obtained/number of bad smell in which refactoring is applied
7. Priority = Sum

Refactoring sequence with the help of greedy algorithm

One-ary analysis
1-ary analysis [RT<=3]

Two-ary analysis
2-ary analysis [RT>=4]
Sequence of Steps for the Proposed Approach

1. **Start**
2. Create 1-ary and 2-ary trees after applying refactoring.
3. Calculate the sum of maintainability at each step.
4. **Is sum minimum?**
   - **Yes**: Move forward in a tree.
   - **No**: Discard the tree from this point.
5. Find the refactoring sequence.
6. **End**
Greedy Algorithm

Greedy algorithm is an algorithmic paradigm that always makes choices that look best at that moment [13][14].

It tries to make locally optimal choices at each stage in hope of finding the globally optimal solution.

We use greedy algorithm is used to find the refactoring sequence for the datasets used.

At every step, we move forward in the tree formed.
We observe that there are classes with number of bad smells more than 5 which clearly shows the need of refactoring and also identifying a correct order of refactoring.

We perform grouping based on the number of bad smells, select classes based on the highest LOC value and then apply the refactoring sequence.

Trees are formed after applying refactoring to the original code in two ways that are discussed below.
Sequence Determination - 8 Classes in JTDS project

Severely affected classes

Grouping based on number of bad smells
- Bad smell=7
  - tdscore.java
- Bad smell=6
  - jtdsstatement.java and sharedsocket.java
- Bad smell=5
  - support.java and BlobBuffer.java
- Bad smell=4
  - ResponseStream.java, jtdsResultSet.java and jtdsDatabaseMetaData.java

Classes in each group
- tdscore.java
- jtdsStatement.java
- support.java
- jtdsDatabaseMetaData.java

Classes selected on the basis of highest LOC value
- tdscore.java LOC=3882
- jtdsStatement.java LOC=1308
- support.java LOC=1306
- jtdsDatabaseMetaData.java LOC=3334

Refactoring applied
- R1, R2, R3, R4, R5
- R1, R3, R4, R5
- R1, R2, R5, R6
- R1, R3, R5
Original Code (OC), Changed Code (CC), Refactoring Technique

EM (Extract Method) and trees such as 1-ary and 2-ary for a class in JTDS
# Table of Contents

1. Research Motivation and Aim
   - Background and Context Setting
   - Research Contributions

2. Related Work

3. Solution Approach
   - N-Ary Trees and Greedy algorithm
   - One-ary analysis
   - Two-ary analysis

4. Dataset and Results
   - Experimental Dataset
   - Experimental Results

5. Conclusion
In one-ary analysis, trees are formed by applying various refactoring techniques on a same portion of the code.

Changed version should be an improved version of the original source code.

After applying the refactoring techniques, this analysis will give the top most refactoring technique.
Four refactoring techniques like Extract Method (EM), Push Up (PU), Push Down (PD) and Move Method (MM) have been applied to the initial original code to get four changed versions of the software.

The CC denotes the changed code and is collaborated along with the refactoring technique in a node of a tree.

Three refactoring techniques need to be eliminated and is done by considering the maintainability value of the software after applying it.
1-ary and 2-ary Tree Representation

1-ary tree

[jtddatabasemetadatad.java]

```
    C
   ↙ ↙
  R1 R3
```

2-ary tree

[Tdscore.java]

```
    C
   ↙ ↙
  R2 R4
       ↙ ↙
   R2 R4 R5
   ↙ ↙ ↙
 R2 R1 R4 R5 R3
```

An Illustration of 1-ary and 2-ary Tree Representation for Classes having Large Number of Bad Smells in JTDS Project
### 8 Classes from *jtds*- Bad Smells Greater than four

<table>
<thead>
<tr>
<th>Classes</th>
<th>Bad Smell (BS)</th>
<th># BS</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 TdsCore.java</td>
<td>God class, long method, type checking, feature envy, empty catch block, exception thrown in finally block, nested try statement</td>
<td>7</td>
<td>3882</td>
</tr>
<tr>
<td>2 JtdsStatement.java</td>
<td>Long method, type checking, feature envy, empty catch block, exception thrown in finally block, nested try statement</td>
<td>6</td>
<td>1308</td>
</tr>
<tr>
<td>3 SharedSocket.java</td>
<td>Feature envy, long method, god class, careless cleanup, empty catch block, exception throw in finally block</td>
<td>6</td>
<td>971</td>
</tr>
<tr>
<td>4 Support.java</td>
<td>Long method, god class, careless cleanup, empty catch block, nested try statement</td>
<td>5</td>
<td>1306</td>
</tr>
<tr>
<td>5 BlobBuffer.java</td>
<td>God class, long method, careless cleanup, empty catch block, exception thrown in finally block</td>
<td>5</td>
<td>1207</td>
</tr>
<tr>
<td>6 ResponseStream.java</td>
<td>God class, long method, feature envy, empty catch block</td>
<td>4</td>
<td>463</td>
</tr>
<tr>
<td>7 JtdsResultSet.java</td>
<td>God class, long method, feature envy, empty catch block</td>
<td>4</td>
<td>1497</td>
</tr>
<tr>
<td>8 JtdsDatabaseMetaData.java</td>
<td>Type checking, long method, empty catch block, exception thrown in finally block</td>
<td>4</td>
<td>3334</td>
</tr>
</tbody>
</table>
Table of Contents

1 Research Motivation and Aim
   • Background and Context Setting
   • Research Contributions

2 Related Work

3 Solution Approach
   • N-Ary Trees and Greedy algorithm
   • One-ary analysis
   • Two-ary analysis

4 Dataset and Results
   • Experimental Dataset
   • Experimental Results

5 Conclusion
Two-ary Analysis - 1

The two-ary trees are formed after combining refactoring techniques on the source code so that percentage of improvement of quality gets increased.

The original code gets converted to the changed code CC1 by applying Extract method class. Afterwards push up refactoring technique is applied to get CC2 and so on.

Different refactoring techniques are applied one after the other on the same portion of the code to get final changed version CC4.
At the end, refactoring sequence is found out to be EM → PU → PD → MM.

The original code gets converted to the changed code CC1 by applying Extract method class. Afterwards push up refactoring technique is applied to get CC2 and so on.

Another Example: There exist two refactoring sequences EM → PU and MM → PD.
Order in which refactoring is applied

Depends on two factors

Presence of a particular bad smell that will help in judging the software developer which particular technique should be used in removing that smell.

Priority of the refactoring techniques, RP is calculated with help of maintainability values and number of classes in which a particular refactoring technique is applied.

Individual refactoring technique $X$ is applied, value of maintainability, $M$ is observed. After dividing $M$ value with number of classes in which $X$ is applied, priority of $X$ technique can be determined.
Table of Contents

1. Research Motivation and Aim
   - Background and Context Setting
   - Research Contributions

2. Related Work

3. Solution Approach
   - N-Ary Trees and Greedy algorithm
   - One-ary analysis
   - Two-ary analysis

4. Dataset and Results
   - Experimental Dataset
   - Experimental Results

5. Conclusion

Sandhya Tarwani, Ashish Sureka
Refactoring Sequence Determination
JTDS and JChess

**JTDS** - It is an open source jdbc 3.0 type 4 driver for Microsoft SQL Server. It is currently the fastest production ready JDBC driver for SQL server. It consists of 64 classes.

**JChess** - It is a java based chess game project that requires two players playing on local computer or via network connection. It consists of 69 classes.

---

\(^a\)https://sourceforge.net/projects/jtds/?source=directory

\(^b\)https://sourceforge.net/projects/jchesslibraryss/?source=directory
Four datasets - sourceforge.net

**OrDrumbox and ArtOfIllusion**

*OrDrumbox*\(^a\) - It is an open source audio sequencer and software drum machine that is used to compose the bass line for completing the song. It consists of 217 classes.

*ArtOfIllusion*\(^b\) - It consists of 739 classes and is a fully featured 3D modeling, rendering and animation studio. It consists of subdivision surface based modeling tools, graphical language for designing etc.

\(^a\)https://sourceforge.net/projects/ordrumbox/?source=directory  
\(^b\)https://sourceforge.net/projects/aoi/?source=directory
Table of Contents

1 Research Motivation and Aim
   • Background and Context Setting
   • Research Contributions

2 Related Work

3 Solution Approach
   • N-Ary Trees and Greedy algorithm
   • One-ary analysis
   • Two-ary analysis

4 Dataset and Results
   • Experimental Dataset
   • Experimental Results

5 Conclusion
Dataset *jtds* consist of 64 classes in which eight classes have been considered as severe as they contain bad smells greater than 4.

The ranks will help in selecting refactoring techniques for the formation of the trees.

Classes are selected on the basis of highest LOC value as higher LOC value will lead towards more confusion and complexities.
### Refactoring Technique, Priority Value and the Rank Assignment

<table>
<thead>
<tr>
<th>Refactoring Technique</th>
<th>Priority Value</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 [EM]</td>
<td>458.0313</td>
<td>5</td>
</tr>
<tr>
<td>R2 [EC]</td>
<td>429.755</td>
<td>3</td>
</tr>
<tr>
<td>R3 [RC]</td>
<td>529.1633</td>
<td>6</td>
</tr>
<tr>
<td>R4 [MM]</td>
<td>345.198</td>
<td>1</td>
</tr>
<tr>
<td>R5 [RE]</td>
<td>431.7525</td>
<td>4</td>
</tr>
<tr>
<td>R6 [TEFB]</td>
<td>385.06</td>
<td>2</td>
</tr>
<tr>
<td>R7 [BOTB]</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>R8 [SC]</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

Sandhya Tarwani, Ashish Sureka

Refactoring Sequence Determination
## Metrics after Applying Refactoring (Type 1 & 2)

<table>
<thead>
<tr>
<th>Sequence</th>
<th>CBO</th>
<th>LCOM</th>
<th>RFC</th>
<th>WMC</th>
<th>Ocavg</th>
<th>AHF</th>
<th>MHF</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM, EM</td>
<td>57</td>
<td>2</td>
<td>122</td>
<td>310</td>
<td>4.84</td>
<td>72.32</td>
<td>73.44</td>
<td>641.6</td>
</tr>
<tr>
<td>MM, RE</td>
<td>57</td>
<td>2</td>
<td>121</td>
<td>340</td>
<td>5.40</td>
<td>72.32</td>
<td>73.02</td>
<td>670.74</td>
</tr>
<tr>
<td>EC, EM</td>
<td>57</td>
<td>5</td>
<td>155</td>
<td>291</td>
<td>4.34</td>
<td>72.73</td>
<td>71.64</td>
<td>656.71</td>
</tr>
<tr>
<td>EC, RE</td>
<td>57</td>
<td>5</td>
<td>153</td>
<td>299</td>
<td>4.60</td>
<td>72.73</td>
<td>70.77</td>
<td>662.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sequence</th>
<th>CBO</th>
<th>LCOM</th>
<th>RFC</th>
<th>WMC</th>
<th>Ocavg</th>
<th>AHF</th>
<th>MHF</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM, EM, RC</td>
<td>57</td>
<td>2</td>
<td>117</td>
<td>297</td>
<td>4.64</td>
<td>72.32</td>
<td>73.44</td>
<td>623.4</td>
</tr>
<tr>
<td>MM, RE, RC</td>
<td>57</td>
<td>2</td>
<td>116</td>
<td>327</td>
<td>5.19</td>
<td>72.32</td>
<td>73.02</td>
<td>652.53</td>
</tr>
<tr>
<td>EC, EM, RC</td>
<td>57</td>
<td>5</td>
<td>150</td>
<td>278</td>
<td>4.15</td>
<td>72.73</td>
<td>71.64</td>
<td>638.52</td>
</tr>
<tr>
<td>EC, RE, RC</td>
<td>57</td>
<td>5</td>
<td>148</td>
<td>286</td>
<td>4.40</td>
<td>72.73</td>
<td>70.77</td>
<td>643.9</td>
</tr>
</tbody>
</table>
Metrics after Applying Refactoring (Type 3 & 4)

<table>
<thead>
<tr>
<th>Sequence</th>
<th>CBO</th>
<th>LCOM</th>
<th>RFC</th>
<th>WMC</th>
<th>Ocavg</th>
<th>AHF</th>
<th>MHF</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM, EM</td>
<td>33</td>
<td>9</td>
<td>103</td>
<td>133</td>
<td>2.56</td>
<td>-167.74</td>
<td>-34.62</td>
<td>78.2</td>
</tr>
<tr>
<td>MM, RC</td>
<td>33</td>
<td>8</td>
<td>100</td>
<td>124</td>
<td>2.48</td>
<td>-167.74</td>
<td>-40</td>
<td>59.74</td>
</tr>
<tr>
<td>RE, EM</td>
<td>33</td>
<td>8</td>
<td>109</td>
<td>151</td>
<td>2.80</td>
<td>-167.74</td>
<td>-37.04</td>
<td>99.02</td>
</tr>
<tr>
<td>RE, RC</td>
<td>33</td>
<td>8</td>
<td>106</td>
<td>142</td>
<td>2.73</td>
<td>-167.74</td>
<td>-42.31</td>
<td>81.68</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sequence</th>
<th>CBO</th>
<th>LCOM</th>
<th>RFC</th>
<th>WMC</th>
<th>Ocavg</th>
<th>AHF</th>
<th>MHF</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC, EM</td>
<td>41</td>
<td>7</td>
<td>29</td>
<td>179</td>
<td>12.79</td>
<td>100</td>
<td>100</td>
<td>468.79</td>
</tr>
<tr>
<td>EC, RE</td>
<td>41</td>
<td>7</td>
<td>27</td>
<td>183</td>
<td>15.25</td>
<td>100</td>
<td>100</td>
<td>473.25</td>
</tr>
<tr>
<td>TEFB, EM</td>
<td>55</td>
<td>11</td>
<td>46</td>
<td>266</td>
<td>9.76</td>
<td>100</td>
<td>100</td>
<td>587.76</td>
</tr>
<tr>
<td>TEFB, RE</td>
<td>55</td>
<td>11</td>
<td>44</td>
<td>270</td>
<td>10.77</td>
<td>100</td>
<td>100</td>
<td>590.77</td>
</tr>
</tbody>
</table>
Refactoring techniques are selected on the basis rank assigned to them.

Result shows the change in value of metrics after applying refactoring techniques so that sum can be calculated for all the metrics that will help in relating it with the maintainability of the class.

If results for tdscore.java are taken into consideration then it can be seen that it is formed by after the application of R2 and R4 refactoring techniques.
Workedout Example - JTDS Dataset

Afterwards, minimum value is selected among all the rows calculation by keeping in mind the inverse relation between the software metrics and maintainability.

Result shows that the combinations of refactoring techniques are applied but this time only that part of the tree is taken forward that gets selected in table.

At the end refactoring sequence for tdscore.java is found to be MM → EM → RC as this combination has minimum value of sum of metrics which results in maximum maintainability.
### Object-Oriented Metrics Value after Applying the Refactoring in the Given Sequence

<table>
<thead>
<tr>
<th>Sequence</th>
<th>CBO</th>
<th>LCOM</th>
<th>RFC</th>
<th>WMC</th>
<th>Ocavg</th>
<th>AHF</th>
<th>MHF</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>EM, RC</td>
<td>10</td>
<td>128</td>
<td>220</td>
<td>267</td>
<td>1.44</td>
<td>100</td>
<td>100</td>
<td>826.44</td>
</tr>
<tr>
<td>RE, RC</td>
<td>10</td>
<td>128</td>
<td>220</td>
<td>275</td>
<td>1.40</td>
<td>100</td>
<td>100</td>
<td>834.4</td>
</tr>
</tbody>
</table>
Example - Refactoring Sequence Determination Algorithm

<table>
<thead>
<tr>
<th>Sequence</th>
<th>CBO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tdscore.java (7 smells)</td>
<td>MM → EMRC</td>
</tr>
<tr>
<td>Jtdsstatement.java (6 smells)</td>
<td>MM → RC</td>
</tr>
<tr>
<td>Support.java (5 smells)</td>
<td>EC → RE</td>
</tr>
<tr>
<td>Jtdsdatabasemetadata.java (4 smells)</td>
<td>EM → RC</td>
</tr>
</tbody>
</table>

Four Classes from JTDS dataset selected as Illustrative Example to Demonstrate Refactoring Sequence Determination Algorithm
The number of god class and long method type of bad smells are high in all the datasets which indicates the maximum use of extract class and extract method refactoring technique.

Most common refactoring sequence will EC-EM.
## Java Class, Bad Smell Present and Refactoring Sequence

<table>
<thead>
<tr>
<th>Java Classes</th>
<th>Bad Smell Present</th>
<th>Refactoring Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Jchess</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 game.java (4 bad smell)</td>
<td>God class, long method, type checking, feature envy</td>
<td>EM → MM</td>
</tr>
<tr>
<td><strong>OrDrumbox</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 RIFF32Reader.java (7 bad smell)</td>
<td>God class, long method, feature envy, empty catch block, dummy handler, careless cleanup, exception thrown in finally block</td>
<td>EC → MM → TEFB</td>
</tr>
<tr>
<td>3 song.java (6 bad smell)</td>
<td>God class, long method, type checking, feature envy, dummy handler, careless cleanup</td>
<td>EM → MM → TEFB</td>
</tr>
<tr>
<td>4 drumkitManager.java (5 bad smell)</td>
<td>God class, long method, dummy handler, nested try statement, careless cleanup</td>
<td>EC → RE</td>
</tr>
<tr>
<td>5 OrTrack.java (4 bad smell)</td>
<td>God class, long method, feature envy, dummy handler</td>
<td>EM → MM</td>
</tr>
<tr>
<td><strong>ArtOfIllusion</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 artOfIllusion.java (7 bad smell)</td>
<td>God class, long method, empty catch block, dummy handler, unprotected main, nested try catch, careless cleanup</td>
<td>EM → TEFB → BOTB</td>
</tr>
<tr>
<td>7 Scene.java (6 bad smell)</td>
<td>God class, long method, type checking, dummy handler, nested try catch, careless cleanup</td>
<td>RC → EC → TEFB</td>
</tr>
<tr>
<td>8 layoutWindow.java (5 bad smell)</td>
<td>God class, long method, type checking, feature envy, empty catch block</td>
<td>MM → EC → RE</td>
</tr>
<tr>
<td>9 TriMeshEditorWindow.java (4 bad smell)</td>
<td>God class, long method, type checking, feature envy</td>
<td>MM → EC</td>
</tr>
<tr>
<td>Feature</td>
<td>JTDS</td>
<td>JCHESS</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
<td>--------</td>
</tr>
<tr>
<td># Classes</td>
<td>64</td>
<td>69</td>
</tr>
<tr>
<td># Changes</td>
<td>37</td>
<td>57</td>
</tr>
<tr>
<td>1 RT</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>2 RT</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3 RT</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>4 RT</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>&gt;5 RT</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Number of Classes and Changes, Number of Classes for which the Number of Refactoring is Greater than a Defined Value
### Number of Classes Having a Pre-Defined Bad Smell

<table>
<thead>
<tr>
<th>Feature</th>
<th>JTDS</th>
<th>JCHESS</th>
<th>ORDRUMBOX</th>
<th>ARTOFILLUSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>God Class</td>
<td>16</td>
<td>22</td>
<td>78</td>
<td>313</td>
</tr>
<tr>
<td>Long Method</td>
<td>26</td>
<td>22</td>
<td>78</td>
<td>390</td>
</tr>
<tr>
<td>Type Checking</td>
<td>5</td>
<td>3</td>
<td>12</td>
<td>111</td>
</tr>
<tr>
<td>Feature Envy</td>
<td>8</td>
<td>4</td>
<td>31</td>
<td>141</td>
</tr>
<tr>
<td>Empty Catch-Block</td>
<td>15</td>
<td>0</td>
<td>7</td>
<td>51</td>
</tr>
<tr>
<td>Dummy Handler</td>
<td>1</td>
<td>10</td>
<td>43</td>
<td>62</td>
</tr>
<tr>
<td>Exception - Finally</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Careless Cleanup</td>
<td>5</td>
<td>1</td>
<td>8</td>
<td>22</td>
</tr>
<tr>
<td>Unprotected Main</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Nested Try</td>
<td>6</td>
<td>1</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>Over Logging</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Consider dataset Artofillusion, 194 classes out of 739 classes will remain unchanged.

Over logging type of bad smell is not present in any dataset and hence sprout class refactoring technique will never be a part of refactoring sequence.
We present an approach to determine the refactoring sequence for a software system having bad smells in several classes.

The proposed approach is based on identifying bad smells for each class in the object-oriented software system (number of bad smells and types of smell) and grouping the classes based on the number of bad smells.

We conduct experiments on four open-source Java projects to demonstrate the effectiveness of our approach.

We present descriptive statistics of the final results which shows that the proposed approach meets its desired objectives.
References I


