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

Sandhya Tarwani[1] Ashish Sureka[2]

[1]SRM University, India (sandhya.tarwani@gmail.com)

[2]Ashoka University, India (ashish.sureka@ashoka.edu.in)

QuASoQ 2017 (co-located to APSEC 2017)

Table of Contents

Table of Contents

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.

Refactoring Sequence Determination

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

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 algoritm.

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

Literature Survey - 1

A. Ghannem et al. [3]

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]

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.

Literature Survey - 2

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.

Literature Survey - 3

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

Dataset and N-Ary Trees

We download source-code dataset from sourceforge and bad smells are identified with the help of plug-ins like JDeodorant [9][12]

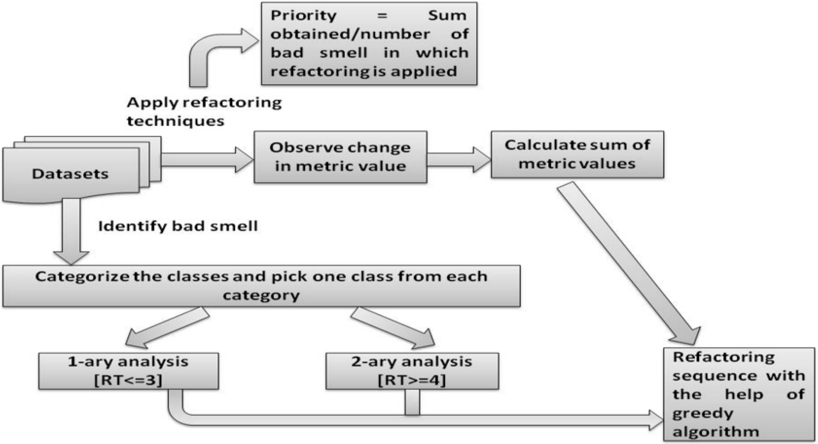
Classes are then prioritized on the basis of number of bad smells

Only those classes are considered whose number of bad smells are greater than or equal to 4

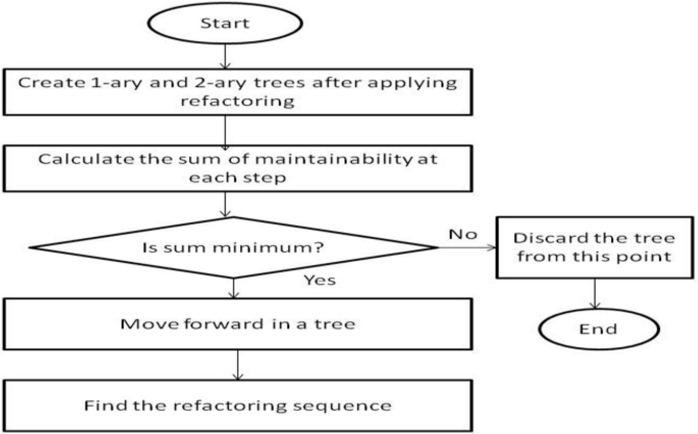
1-ary tree is formed where number of refactoring techniques is less than or equal to 3 and 2-ary tree is formed otherwise.

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



Sequence of Steps for the Proposed Approach



Dataset and N-Ary Trees

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.

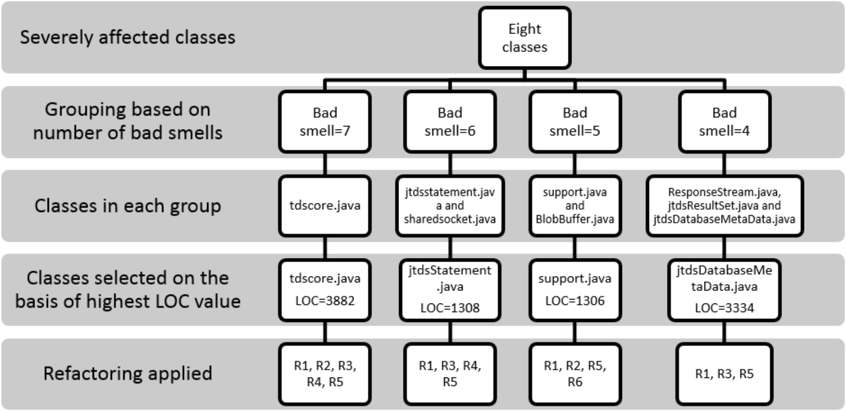
Grouping based on the Number of Bad Smells

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.

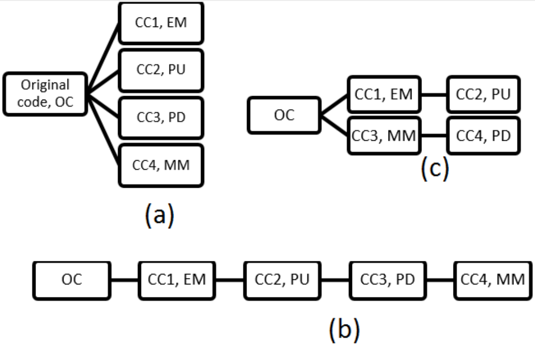
We perform grouping based on the number of bad smells, select classes based on the highest LOC value and then apply the refactoring sequence.

Trees are formed after applying refactoring to the original code in two ways that are discussed below.

Sequence Determination - 8 Classes in JTDS project



Original Code (OC), Changed Code (CC), Refactoring Technique



EM (Extract Mehtod) and trees such as 1-ary and 2-ary for a class in JTDS

Table of Contents

One-ary Analysis - 1

In one-ary analysis, trees are formed by applying various refactoring techniques on a same potion 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

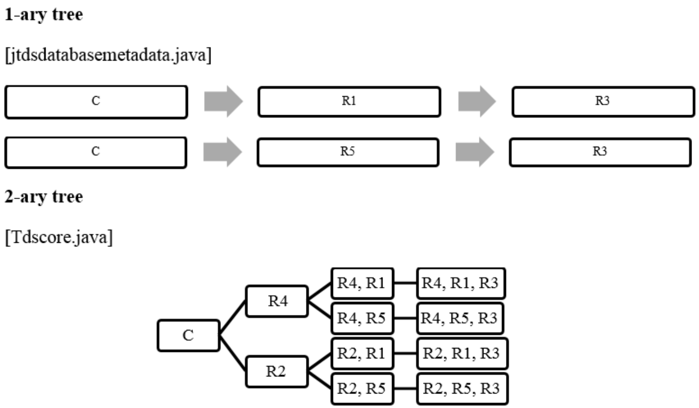
One-ary Analysis - 2

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



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

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

Table of Contents

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.

Two-ary Analysis - 2

At the end, refactoring sequence is found out to be EM $\rightarrow$ PU $\rightarrow$ PD $\rightarrow$ 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 $\rightarrow$ PU and MM $\rightarrow$ 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

Four datasets - sourceforge.net

JTDS and JChess

JTDS[a] - 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[b] - 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

Workedout Example - JTDS Dataset

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

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

Metrics after Applying Refactoring (Type 1 & 2)

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

Sequence	CBO	LCOM	RFC	WMC	Ocavg	AHF	MHF	Total
MM, EM, RC	57	2	117	297	4.64	72.32	73.44	623.4
MM, RE, RC	57	2	116	327	5.19	72.32	73.02	652.53
EC, EM, RC	57	5	150	278	4.15	72.73	71.64	638.52
EC, RE, RC	57	5	148	286	4.40	72.73	70.77	643.9

Metrics after Applying Refactoring (Type 3 & 4)

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

Sequence	CBO	LCOM	RFC	WMC	Ocavg	AHF	MHF	Total
EC, EM	41	7	29	179	12.79	100	100	468.79
EC, RE	41	7	27	183	15.25	100	100	473.25
TEFB, EM	55	11	46	266	9.76	100	100	587.76
TEFB, RE	55	11	44	270	10.77	100	100	590.77

Workedout Example - JTDS Dataset

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 $\rightarrow$ EM $\rightarrow$ RC as this combination has minimum value of sum of metrics which results in maximum maintainability.

Metrics Value after Applying the Refactoring

Sequence	CBO	LCOM	RFC	WMC	Ocavg	AHF	MHF	Total
EM, RC	10	128	220	267	1.44	100	100	826.44
RE, RC	10	128	220	275	1.40	100	100	834.4

Object-Oriented Metrics Value after Applying the Refactoring in the Given Sequence

Example - Refactoring Sequence Determination Algorithm

Sequence	CBO
Tdscore.java (7 smells)	MM $\rightarrow$ EMRC
Jtdsstatement.java (6 smells)	MM $\rightarrow$ RC
Support.java (5 smells)	EC $\rightarrow$ RE
Jtdsdatabasemetadata.java (4 smells)	EM $\rightarrow$ RC

Four Classes from JTDS dataset selected as Illustrative Example to Demonstrate Refactoring Sequence Determination Algorithm

Analysis across Datasets

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

	Java Classes	Bad Smell Present	Refacotring Sequence
		Jchess	
1	game.java (4 bad smell)	God class, long method, type checking, feature envy	EM → MM
		OrDrumbox	
2	RIFF32Reader.java (7 bad smell)	God class, long method, feature envy, empty catch block, dummy handler, careless cleanup, exception thrown in finally block	EC → MM → TEFB
3	song.java (6 bad smell)	God class, long method, type checking, feature envy, dummy handler, careless cleanup	EM → MM → TEFB
4	drumkitManager.java (5 bad smell)	God class, long method, dummy handler, nested try statement, careless cleanup	EC → RE
5	OrTrack.java (4 bad smell)	God class, long method, feature envy, dummy handler	EM → MM
		ArtOfIllusion	
6	artOfIllusion.java (7 bad smell)	God class, long method, empty catch block, dummy handler, unprotected main, nested try catch, careless cleanup	EM → TEFB → BOTB
7	Scene.java (6 bad smell)	God class, long method, type checking, dummy handler, nested try catch, careless cleanup	RC → EC → TEFB
8	layoutWindow.java (5 bad smell)	God class, long method, type checking, feature envy, empty catch block	MM → EC → RE
9	TriMeshEditorWindow.java (4 bad smell)	God class, long method, type checking, feature envy	MM → EC

Result - Values Greater than Defined Value

Feature	JTDS	JCHESS	ORDRUMBOX	ARTOFILLUSION
# Classes	64	69	217	739
# Changes	37	57	86	194
1 RT	8	8	49	248
2 RT	6	2	44	144
3 RT	6	0	25	75
4 RT	3	2	7	39
>5 RT	4	0	6	39

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

Feature	JTDS	JCHESS	ORDRUMBOX	ARTOFILLUSION
God Class	16	22	78	313
Long Method	26	22	78	390
Type Checking	5	3	12	111
Feature Envy	8	4	31	141
Empty Catch-Block	15	0	7	51
Dummy Handler	1	10	43	62
Exception - Finally	5	0	1	4
Careless Cleanup	5	1	8	22
Unprotected Main	1	3	3	3
Nested Try	6	1	5	15
Over Logging	0	0	0	0

Analysis across Datasets

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.

Summary and Takeaways

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

[1] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[2] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[3] A. Ghannem, G. El Boussaidi, and M. Kessentini. Model refactoring using interactive genetic algorithm. In *International Symposium on Search Based Software Engineering*, pages 96–110. Springer, 2013.

[4] A.-R. Han and D.-H. Bae. An efficient method for assessing the impact of refactoring candidates on maintainability based on matrix computation. In *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, volume 1, pages 430–437. IEEE, 2014.

[5] Y. Khrishe and M. Alshayeb. An empirical study on the effect of the order of applying software refactoring. In *Computer Science and Information Technology (CSIT), 2016 7th International Conference on*, pages 1–4. IEEE, 2016.

[6] P. Meananeatra. Identifying refactoring sequences for improving software maintainability. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 406–409. ACM, 2012.

References II

[7] E. Piveta, J. Araújo, M. Pimenta, A. Moreira, P. Guerreiro, and R. T. Price. Searching for opportunities of refactoring sequences: reducing the search space. In *Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International*, pages 319–326. IEEE, 2008.

[8] A. Shahjahan, W. haider Butt, and A. Z. Ahmad. Impact of refactoring on code quality by using graph theory: An empirical evaluation. In *SAI Intelligent Systems Conference (IntelliSys), 2015*, pages 595–600. IEEE, 2015.

[9] G. Szőke, C. Nagy, L. J. Fülöp, R. Ferenc, and T. Gyimóthy. Faultbuster: An automatic code smell refactoring toolset. In *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pages 253–258. IEEE, 2015.

[10] S. Tarwani and A. Chug. Sequencing of refactoring techniques by greedy algorithm for maximizing maintainability. In *Advances in Computing, Communications and Informatics (ICACCI), 2016 International Conference on*, pages 1397–1403. IEEE, 2016.

References III

[11] I. Toyoshima, S. Yamaguchi, and J. Zhang. A refactoring algorithm of workflows based on petri nets. In *Advanced Applied Informatics (IIAI-AAI), 2015 IIAI 4th International Congress on*, pages 79–84. IEEE, 2015.

[12] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 329–331. IEEE, 2008.

[13] R. Wongpiang and P. Muenchaisri. Selecting sequence of refactoring techniques usage for code changing using greedy algorithm. In *Electronics Information and Emergency Communication (ICEIEC), 2013 IEEE 4th International Conference on*, pages 160–164. IEEE, 2013.

[14] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning dna sequences. *Journal of Computational biology*, 7(1-2):203–214, 2000.