Analyzing Code Smell Removal Sequences for Enhanced Software Maintainability

A Presentation
on
M.Tech Dissertation Work

Supervised By
Dr. Paramvir Singh
Assistant Professor
Department of CSE, NITJ

Supervised By
Dr. Ashish Sureka
Principal Research Scientist
ABB INCRC, Bangalore

Presented By
Yukti Mehta
M.Tech II Year, IS
Roll No- 15223003

Table of Contents

Introduction

- In modern day software development, maintenance and evolution are the keys for a successful software product.
- During the evolution of a software, its design becomes more complicated; hence reducing the quality of the software.
- Maintenance activities account for about 90% of the total project cost.
- While developing the code, developers are required to enhance its quality—for example, maintainability.
- Most of the companies incur huge losses for scarping off projects when they fail to meet the completion deadlines.
- Researchers and developers are looking for solutions to obtain maintainable software with less effort, time and cost.

Introduction (Contd.)

- **Code Smells**
  - Any symptoms in the source code of a program that indicate deeper problems.
  - These smells adversely influence framework lifecycle properties, such as testability, understandability, reusability and extensibility.
  - 22 code smells are defined by Fowler.
  - Examples of code smells incorporate *duplicate code*, *long method*, *god class*, *feature envy* and *long parameter lists*.
- **Code Smell Detection Tools**

Tool	Type	Supported Languages	Automated Refactoring	Direct Link to Code
Checkstyle	Eclipse Plugin	Java	No	No
Décor	Standalone	Java	No	No
Iplasma	Standalone	C++, Java	No	No
Infusion	Standalone	C, C++, Java	No	No
JDeodorant	Eclipse Plugin	Java	Yes	Yes

Introduction (Contd.)

- **Refactoring**
  - Improves the internal design structure of the object oriented software while maintaining its external behavior.
  - Exposes simpler components which are used in making complex expressions.
- **Refactoring Activities**
  - Determine the code fragment where refactoring should be applied.
  - Determine the appropriate refactoring(s) to remove the identified code smell.
  - Assure that the applied refactoring preserves software behavior.
  - Apply the refactoring.
  - Determine the impact of refactoring on quality attributes of software.
  - Maintain the consistency among refactored source code and various software artifacts.

Introduction (Contd.)

- **Software Maintainability Metrics**
  - Functions which are used to measure the characteristics of source code such as number of lines in a source code, number of methods, number of classes, coupling between objects etc.
  - Useful as they help tracking down the risks and those code sections that can be troublesome in future.
- **Maintainability Metrics Considered**
  - **Lines of Code (LOC):** The metrics LOC defines the number of lines in the source code after excluding comment lines or blank lines.
  - **Number of Classes (NOCl):** This metric computes the total count of classes present in the source code.
  - **Cyclomatic Complexity Number (v(G)):** This metric computes the complexity of source code and is used in white box testing. It calculates the number of independent paths in a program.
  - **Maintainability Index (MI):** Maintainability Index measures how maintainable (easy to change and support) the source code is.
  - **Relative Logical Complexity (RLC):** This metric is defined as the number of binary decisions divided by the number of statements.

Effect of Code Smells on Maintainability

- Code smells are signs of design issues that can lower code maintainability.
- The existence of code smells seems to be an ideal indicator to assess the maintainability.
- Professional software developers have not explored this assumption in controlled studies.
- As per Fowler, design problems appear as "bad smells" at design or code level that can be eliminated by applying appropriate refactoring.
- As various code smells can be automatically detected, it is appealing to evaluate their scope to expose factors that affect maintainability.

Need of Prioritization to Achieve Maintainable Software

- Developers face constant pressure to spend most of their time to add new features instead of refactoring the code.
- Major hurdles in adopting refactoring in industrial projects have been listed by various researchers.
- These are fear of breaking code, getting management buy-in, deadline pressure, lack of awareness and inadequate support.
- In large-scale systems, the number of code-smells to fix can be quite large in number and it becomes difficult to fix all of them automatically.
- Thus, eventually the prioritization of the code-smells becomes important.
- It can be done based on different criteria such as the importance and risk of classes.

Need of Prioritization to Achieve Maintainable Software (Contd.)

- **Prioritization of Code Smells**
  - Fowler has introduced 22 code smells, but there is a lack of guidance from him as he did not throw light on prioritization of code smells.
  - It is challenging for developers to decide which code smell should be refactored first.
  - The number of smells returned as output by these tools generally exceed the amount of problems which can be dealt by the developer.
  - Researchers proposed various techniques for code smell prioritization based on the impact of the code smells on the overall software quality.
  - A tool is developed that suggests code smells ranking.
  - Some researchers introduced approaches that consider the removal of such code smells that involves low refactoring cost and are easier to refactor.

Need of Prioritization to Achieve Maintainable Software (Contd.)

- **Prioritization of Code Smell Refactorings**
  - To enhance the maintainability of the software, several refactoring techniques to the source code are applied.
  - Different refactoring selection leads to variations in modified code components and different levels of software quality.
  - As per the 2011 survey, four main criteria to select an optimal refactoring sequence considered by developers include:
    - The number of removed bad smells
    - Maintainability
    - Number of modified program elements
    - The size of refactoring sequence
  - Greedy algorithm can be applied to determine the most appropriate sequences of refactoring techniques .

Need of Prioritization to Achieve Maintainable Software (Contd.)

- **Prioritization of Classes in need of Refactoring**
  - An approach is required to identify and prioritize object oriented software classes that are in urgent need of refactoring.
  - It is not possible to refactor every smelly class in the software with limited resources and budget.
  - Prioritizing the critically affected classes assists developers in locating the software portions that need urgent refactoring treatments.
  - Some researchers utilize static metrics like size, complexity and maintainability of the class to categorize the critical classes.
  - The object oriented design metrics of the class also indicate its fault-proneness.
  - Some researchers used class rank prioritization approach to identify the most refactoring-prone as well as architecturally relevant classes by generating the class ranks.

Literature Review

Author	Year	Title	Approach
Rasool et al. [7]	2015	A review of code smell mining techniques	Classification of selected code smell detection techniques and tools based on their detection methods
Palomba et al. [4]	2015	Textual analysis for code smell detection	Proposed *TACO* (Textual Analysis for Code smell detectiOn) for detecting the Long Method smell
Choudhary et al. [14]	2016	Minimizing Refactoring Effort through Prioritization of Classes based on Historical, Architectural and Code Smell Information	Identified the most refactoring-prone as well as architecturally relevant classes and then generated class ranks based on the code smell information.
Tarwani et al. [10]	2016	Sequencing of refactoring techniques by Greedy algorithm for maximizing maintainability	Identified sequence for refactoring as well as best refactoring which will increase maintainability and enhance software quality.
Dhaka et al. [19]	2016	An Empirical Investigation into Code Smell Elimination Sequences for Energy Efficient Software	Empirically investigated the impact of eliminating a set of three code smells, individually as well as in all six possible sequences, on energy consumption behavior of software systems.

Literature Review (Contd.)

Author	Year	Title	Approach
Wongpiang et al. [5]	2013	Selecting sequence of refactoring techniques usage for code changing using greedy algorithm	Proposed an approach for selecting sequence of refactoring techniques usage for code changing using Greedy Algorithm.
Yamashita et al. [16]	2012	Do code smells reflect important maintainability aspects?	Investigated the extent to which code smells reflect factors affecting maintainability that have been identified as important by programmers.
Ouni et al. [9]	2015	Prioritizing code-smells correction tasks using chemical reaction optimization.	Proposed an approach based on a chemical reaction optimization metaheuristic search to find the suitable refactoring solutions.
Sharma et al. [2]	2015	Challenges to and Solutions for Refactoring Adoption: An Industrial Perspective	Highlighted common challenges to refactoring adoption and outlined ways to address these challenges
Olbrich et al. [8]	2010	Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems	Investigated the claim that classes that are involved in certain code smells are liable to be changed more frequently and have more defects than other classes in the code.
Malhotra et al. [15]	2015	Prioritization of classes for refactoring: A step towards improvement in software quality	Proposed a framework to identify the potential classes which immediately require refactoring based on the bad smells as well as design characteristics.

Literature Review (Contd.)

Author	Year	Title	Approach
Marinescu et al. [3]	2004	Detection strategies: Metrics-based rules for detecting design flaws	Proposed a detection strategy for formulating metrics-based rules that capture deviations from good design principles and heuristics.
Tsantalis et al. [20]	2008	JDeodorant: Identification and removal of type-checking bad smells	Presented an Eclipse plug-in that automatically identifies Type-Checking bad smells in Java source code and resolves them.
Mansoor et al. [6]	2013	Code-smells detection using good and bad software design examples	Used multi-objective genetic programming (MOGP) to find the best combination of metrics that maximizes the detection of code-smells.
Frappier et al. [22]	1994	Software metrics for predicting maintainability	Identified a set of software measures based on correlations published in empirical studies.
Fowler et al. [1]	1999	Refactoring: improving the design of existing code	Showed how to do refactoring in a controlled and efficient manner.
Castillo et al. [11]	2014	Analyzing the harmful effect of god class refactoring on power consumption	Analyzed the relationship between architecture sustainability and maintainability by providing empirical evidence of how power consumption increases after refactoring.

Literature Review (Contd.)

Author	Year	Title	Approach
Anda et al. [12]	2007	Assessing software system maintainability using structural measures and expert assessments	Described an empirical study in which the maintainability of four functionally equivalent systems developed in Java was assessed using both structural measures and expert assessments.
Peters et al. [17]	2012	Evaluating the lifespan of code smells using software repository mining	Investigated the lifespan of code smells and the refactoring behaviour of developers in seven open source systems.
Piveta et al. [21]	2008	Searching for opportunities of refactoring sequences: reducing the search space.	Proposed an approach to narrow the number of refactoring sequences by discarding those that semantically does not make sense and avoiding those that lead to the same results.
Travasoos et al. [18]	1999	Detecting defects in object-oriented designs: using reading techniques to increase software quality	Assessed the feasibility of new reading techniques, and discussed the changes made to the latest version of the techniques based on the results of this study.
Vidal et al. [13]	2016	An approach to prioritize code smells for refactoring	Developed a tool that suggests a ranking of code smells, based on a combination of three criteria, namely: past component modifications, important modifiability scenarios for the system, and relevance of the kind of smell.
Meananeatra et al. [23]	2012	Identifying refactoring sequences for improving software maintainability	Proposed an approach to identify an optimal refactoring sequence that meets 2011 survey criteria.

Motivations

- Software maintenance task is very expensive and tedious.
- Maintaining the software is a broad activity which includes correcting the errors, enhancing the capabilities, and optimization.
- Hence, it is required to reduce the effort spent on software maintenance so that the overall cost of the software project can be reduced.
  - During literature survey it is found that an approach is proposed which identifies and prioritizes object oriented software classes that are in need of refactoring.
  - An approach has also been proposed to evaluate refactoring sequence by using the greedy algorithm.
  - The impact of removing code smells individually and sequentially has also been analysed by presenting an empirical relationship between maintainability and sustainability (in terms of energy).
  - The impact of removing *god classes* on two open source software applications in terms of energy consumption using JDeodorant has also been considered.
- Such literature and the literature similar to it constituted the motivation to exploit the optimal code removal sequence that results in a software version with highest maintainability.

Research Objectives

- As a part of this dissertation work, following objectives are framed:

- **Objective 1:**
  - To find whether removing code smells in different permutations yields different software maintainability.

- **Objective 2:**
  - To empirically find the code smell removal sequences that provide the most maintainable software.

Proposed Approach

- For performing experiments, we considered 16 open source Java applications.

Java Applications with their Set #, NOCI and LOC

Java Project	Set #	NOCI	LOC
Spinfo	I	12	1373
Algorithms	I	17	620
Frogger	I	20	1283
JavaShooterGame	I	23	2378
JavaBoatGame	I	30	3483
TigerIsland	I	46	5564
JavaGame	I	55	3673
Jpacman	I	55	2443
Jadventure	I	60	4925
Javaanpr	I	66	4782
CommonBeans	II	111	11734
Pagseguro	II	132	9075
GnuBridge	II	160	11362
Dungeon	II	197	9988
FernFlower	II	225	28989
Jsmpp	II	307	18873

Proposed Approach (Contd.)

- **Selection of Java Open Source Projects and Eclipse Plugins**
  - We cloned the git repositories with Eclipse.
  - To identify design problems, known as bad smells, and resolve them by applying appropriate refactorings, an eclipse plug-in JDeodorant is used.
  - Set I contains the open source java projects which contain less than 100 classes and Set II consists of open source java projects with classes in the range of 100 to 400.
- **Choosing the Maintainability Predicting Metrics**
- After doing research on the metrics, we choose two software metrics, Maintainability Index (MI) and relative logical complexity (RLC).
- A high value of maintainability index means less effort is required to maintain the code.
- The maintainability of the software decreases as the value of RLC increases.
- The proposed new metric, Maintainability Complexity Index (MCI) is calculated using the above two mentioned metrics.

$$MCI = MI * RLC$$

Proposed Approach (Contd.)

- **Prioritization of Software Class Files**
  - In large-scale systems, the number of code smell instances to fix can be quite large and to fix all of them automatically becomes tedious.
  - Thus, the prioritization of the list of code-smells is important.
  - The above proposed metric, MCI, is used to identify the potential classes that are in the need of immediate refactoring.
  - Sort the classes from smallest to largest values.
- **Way of Treating the Code Smells**
  - We use JDeodorant tool which automatically detects the code smells and suggests the best suited refactoring technique.
  - The classes in the original version (OV) of the application are analysed in terms of metrics, MI and RLC.
  - The list is sorted as the class having minimum value of MCI comes first
  - This list serves as the prioritized version (PV).
  - As we are examining the three decisive code smells, six possible permutations can be formed.
  - The sequence in which the code smells are to be eliminated is decided.
  - All the instances of a particular code smell are removed based on the PR of the project which forms Initial Refactored Version (IRV).
  - After that, another code smell is selected and all the instances are removed and then finally the instances of the last code smell left are eliminated which forms the Final Refactored Version (FRV).
  - Each project under examination generates 6 final refactored versions.
  - FRVs of the java application Frogger include Frogger-LM-GC-FE, Frogger-LM-FE-GC, Frogger-GC-LM-FE, Frogger-GC-FE-LM, Frogger-FE-LM-GC, and Frogger-FE-GC-LM.
  - We get 16 IRVs and the count of FRV goes to 96 (16 * 6) of the subjects under study.

Methodology and Implementation

Java Applications and the Number of Code Smell Instances

Java Project	#LM	#GC	#FE
Spinfo	47	3	1
Algorithms	29	6	3
Frogger	36	11	1
JavaShooterGame	50	2	1
JavaBoatGame	29	25	2
TigerIsland	78	22	14
JavaGame	37	16	3
Jpacman	35	17	7
Jadventure	60	6	9
Javaanpr	138	12	3
CommonBeans	111	24	1
Pagseguro	77	10	1
GnuBridge	195	23	5
Dungeon	93	22	18
FernFlower	2	30	14
Jsmpp	100	13	1

Methodology and Implementation (Contd.)

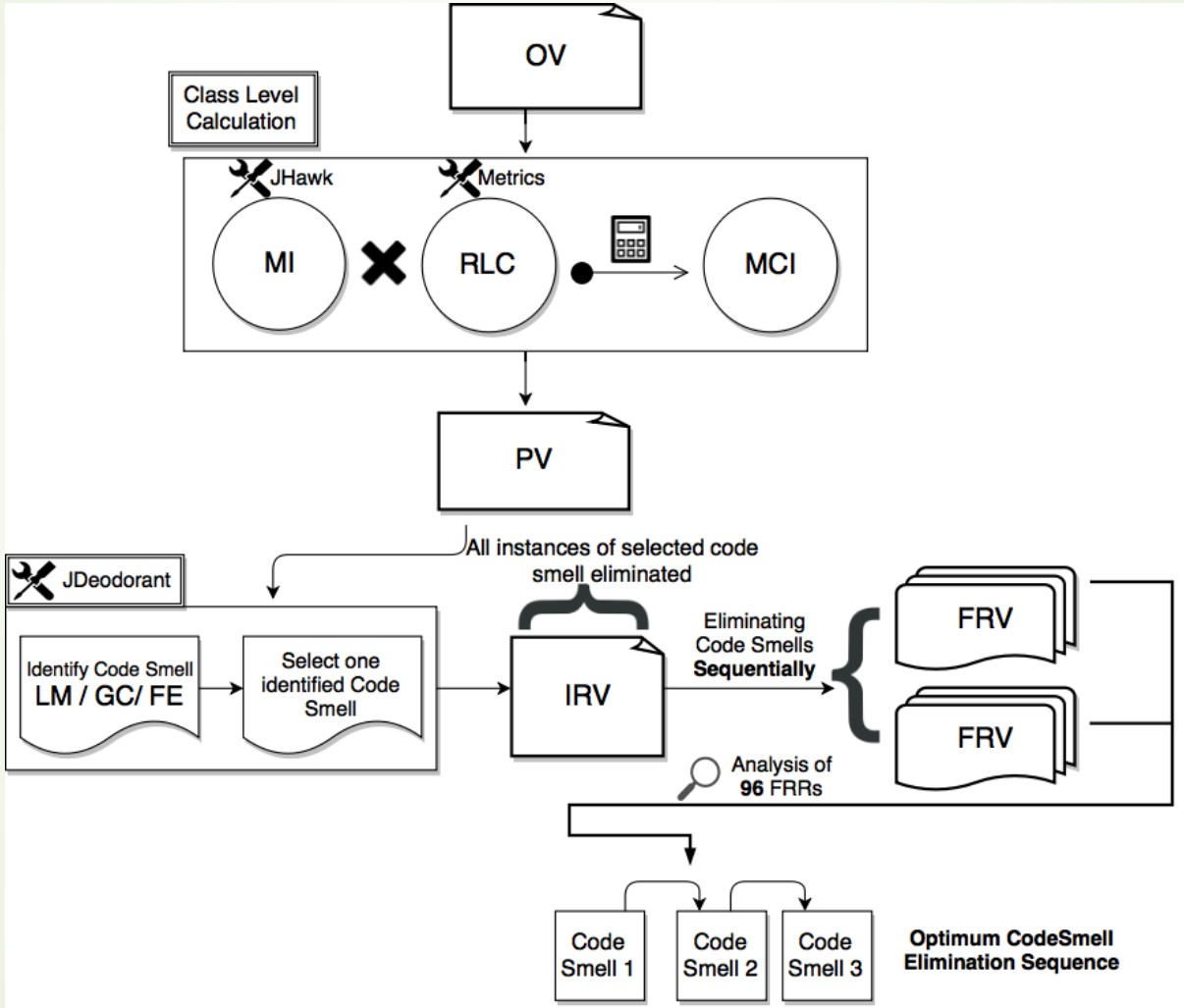


Illustration of Methodology Flow

Tools Used

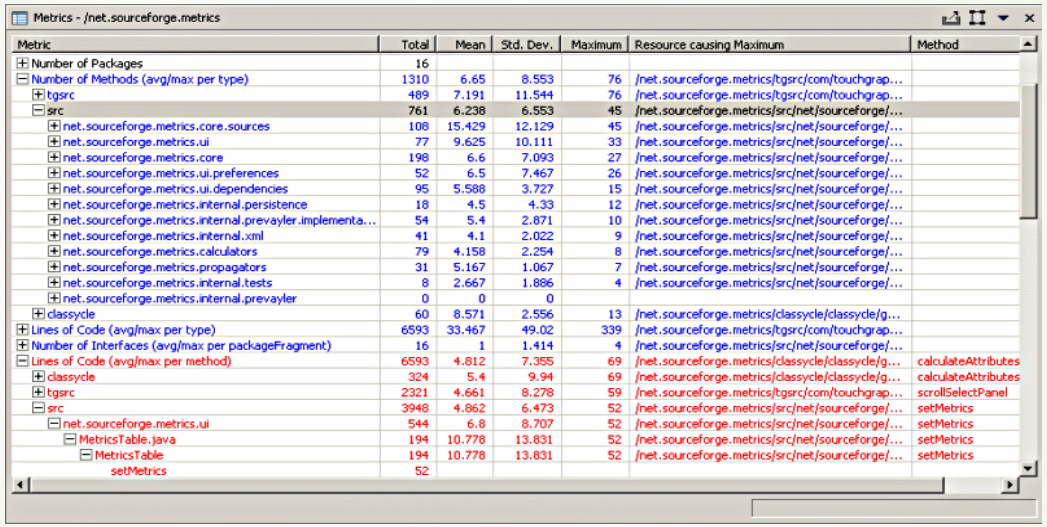
- **JDeodorant**
  - JDeodorant exercises several novel techniques to find out and eliminate code smells by applying appropriate refactoring techniques.
  - JDeodorant is an eclipse plugin.

Refactoring Techniques Applied by JDeodorant on Various Code Smells

Code Smell	Refactoring
Feature Envy	Move Method
Type Checking	Replace Type code with State/Strategy and Replace Conditional with Polymorphism
Long Method	Extract Method
God Class	Extract Class
Duplicated Code	Extract Clone

Tools Used (Contd.)

- **Metrics**
  - Metrics plug in calculates various metrics for the code during build cycle.


Metrics - /net.sourceforge.metrics

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
Number of Packages	16					
Number of Methods (avg/max per type)	1310	6.65	8.553	76	/net.sourceforge.metrics/tgsrc/com/touchgrap...	
tgsrc	489	7.191	11.544	76	/net.sourceforge.metrics/tgsrc/com/touchgrap...	
src	761	6.238	6.553	45	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.core.sources	108	15.429	12.129	45	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.ui	77	9.625	10.111	33	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.core	198	6.6	7.093	27	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.ui.preferences	52	6.5	7.467	26	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.ui.dependencies	95	5.588	3.727	15	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.persistence	18	4.5	4.33	12	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.prevayler.implementa...	54	5.4	2.871	10	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.xml	41	4.1	2.022	9	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.calculators	79	4.158	2.254	8	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.propagators	31	5.167	1.067	7	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.tests	8	2.667	1.886	4	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.prevayler	0	0	0			
classycle	60	8.571	2.556	13	/net.sourceforge.metrics/classycle/classycle/g...	
Lines of Code (avg/max per type)	6593	33.467	49.02	339	/net.sourceforge.metrics/tgsrc/com/touchgrap...	
Number of Interfaces (avg/max per packageFragment)	16	1	1.414	4	/net.sourceforge.metrics/src/net/sourceforge/...	
Lines of Code (avg/max per method)	6593	4.812	7.355	69	/net.sourceforge.metrics/classycle/classycle/g...	calculateAttributes
classycle	324	5.4	9.94	69	/net.sourceforge.metrics/classycle/classycle/g...	calculateAttributes
tgsrc	2321	4.661	8.278	59	/net.sourceforge.metrics/tgsrc/com/touchgrap...	scrollSelectPanel
src	3948	4.862	6.473	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
net.sourceforge.metrics.ui	544	6.8	8.707	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
MetricsTable.java	194	10.778	13.831	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
MetricsTable	194	10.778	13.831	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
setMetrics	52					

Metrics Calculation Progress indicated by Metrics View

- **JHawk**
  - JHawk is a static code analysis tool.
  - It takes the source code of the project and calculates metrics based on numerous aspects of the code - for example complexity, volume, relationships between packages and class and relationships within classes and packages.

Experimental Results

- **Metric Values for Subject Systems**
  - Following tables show the results of the analysed projects

Metric Values for Spinfo and its Refactored Versions

Java Project	NOCI	LOC	MI	RLC	MCI	CC
Spinfo	12	1373	140.46	0.048	6.74208	1.686
$Spinfo_{LM-GC-FE}$	17	1398	141.78	0.060	8.5068	1.65
$Spinfo_{LM-FE-GC}$	16	1406	142.29	0.060	8.5374	1.66
$Spinfo_{GC-LM-FE}$	17	1410	142.29	0.0617	8.779293	1.66
$Spinfo_{GC-FE-LM}$	17	1411	141.89	0.0616	8.740424	1.67
$Spinfo_{FE-LM-GC}$	15	1399	142.25	0.0607	8.634575	1.65
$Spinfo_{FE-GC-LM}$	16	1387	142.15	0.0612	8.69958	1.65

Metric Values for Algorithms and its Refactored Versions

Java Project	NOCI	LOC	MI	RLC	MCI	CC
Algorithms	17	620	80.77	0.095	7.67315	1.68
$Algorithms_{LM-GC-FE}$	21	868	103.16	0.070	7.2212	1.829
$Algorithms_{LM-FE-GC}$	21	870	103.16	0.070	7.2212	1.831
$Algorithms_{GC-LM-FE}$	20	877	103.26	0.067	6.91842	1.821
$Algorithms_{GC-FE-LM}$	20	878	103.26	0.067	6.91842	1.823
$Algorithms_{FE-LM-GC}$	21	865	103.31	0.069	7.12839	1.823
$Algorithms_{FE-GC-LM}$	21	860	103.87	0.07	7.2709	1.823

Experimental Results (Contd.)

Metric Values for Frogger and its Refactored Versions

Java Project	NOCI	LOC	MI	RLC	MCI	CC
Frogger	20	1283	114.74	0.123	14.11302	1.859
$Frogger_{LM-GC-FE}$	30	1613	114.75	0.1177	13.50608	1.796
$Frogger_{LM-FE-GC}$	30	1613	114.71	0.1171	13.43254	1.815
$Frogger_{GC-LM-FE}$	29	1600	114.74	0.11875	13.62538	1.808
$Frogger_{GC-FE-LM}$	29	1620	114.75	0.1126	12.92085	1.835
$Frogger_{FE-LM-GC}$	28	1555	114.61	0.1125	12.89363	1.823
$Frogger_{FE-GC-LM}$	27	1528	114.82	0.1191	13.67506	1.854

Metric Values for JavaShooterGame and its Refactored Versions

Java Project	NOCI	LOC	MI	RLC	MCI	CC
JSG	23	2378	52.14	0.058	3.02412	3.178
$JSG_{LM-GC-FE}$	24	2454	65.58	0.0599	3.928242	3.017
$JSG_{LM-FE-GC}$	27	2578	80.33	0.0655	5.261615	2.893
$JSG_{GC-LM-FE}$	24	2447	65.47	0.0592	3.875824	3.04
$JSG_{GC-FE-LM}$	24	2451	65.52	0.0595	3.89844	3.028
$JSG_{FE-LM-GC}$	24	2451	65.52	0.0595	3.89844	3.028
$JSG_{FE-GC-LM}$	24	2451	65.52	0.0595	3.89844	3.028

Experimental Results (Contd.)

Metric Values for JavaBoatGame and its Refactored Versions

Java Project	NOCl	LOC	MI	RLC	MCI	CC
JBG	30	3483	103.86	0.0315	1.395	1.395
$JBG_{LM-GC-FE}$	43	3938	110.34	0.0286	1.303	1.303
$JBG_{LM-FE-GC}$	44	3968	110.34	0.02872	1.298	1.298
$JBG_{GC-LM-FE}$	42	3934	109.89	0.02872	1.303	1.303
$JBG_{GC-FE-LM}$	43	3958	108.98	0.029	1.302	1.302
$JBG_{FE-LM-GC}$	43	3942	108.67	0.028	1.302	1.302
$JBG_{FE-GC-LM}$	44	3979	108.57	0.028	1.302	1.302

Metric Values for TigerIsland and its Refactored Versions

Java Project	NOCl	LOC	MI	RLC	MCI	CC
TigerIsland	46	5564	98.93	0.0138	1.365234	1.901
$TigerIsland_{LM-GC-FE}$	75	7023	110.61	0.0112	1.238832	1.596
$TigerIsland_{LM-FE-GC}$	76	7026	110.78	0.0111	1.229658	1.592
$TigerIsland_{GC-LM-FE}$	65	6784	110.5	0.0113	1.24865	1.593
$TigerIsland_{GC-FE-LM}$	65	6811	111.75	0.0113	1.262775	1.581
$TigerIsland_{FE-LM-GC}$	76	7018	110.71	0.01114	1.233309	1.59
$TigerIsland_{FE-LM-GC}$	67	6880	109.6	0.01119	1.226424	1.603

Experimental Results (Contd.)

Metric Values for JavaGame and its Refactored Versions

Java Project	NOCl	LOC	MI	RLC	MCI	CC
JavaGame	55	3673	114.74	0.043	4.93382	1.859
$JavaGame_{LM-GC-FE}$	69	4326	110.91	0.0423	4.691493	1.743
$JavaGame_{LM-FE-GC}$	69	4339	114.71	0.0435	4.989885	1.762
$JavaGame_{GC-LM-FE}$	67	4302	114.74	0.044	5.04856	1.754
$JavaGame_{GC-FE-LM}$	66	4236	114.82	0.0429	4.925778	1.78
$JavaGame_{FE-LM-GC}$	70	4312	114.61	0.040	4.5844	1.769
$JavaGame_{FE-GC-LM}$	66	4306	114.82	0.042	4.82244	1.8

Metric Values for Jpacman and its Refactored Versions

Java Project	NOCl	LOC	MI	RLC	MCI	CC
Jpacman	55	2443	101.32	0.0184	1.864288	1.361
$Jpacman_{LM-GC-FE}$	61	2697	127.8	0.0137	1.75086	1.306
$Jpacman_{LM-FE-GC}$	60	2678	127.79	0.01344	1.717498	1.308
$Jpacman_{GC-LM-FE}$	61	2701	127.8	0.0136	1.73808	1.301
$Jpacman_{GC-FE-LM}$	61	2698	127.8	0.0137	1.75086	1.302
$Jpacman_{FE-LM-GC}$	59	2668	127.79	0.01349	1.723887	1.31
$Jpacman_{FE-GC-LM}$	55	2557	127.8	0.0168	2.14704	1.332

Experimental Results (Contd.)

Metric Values for Jadventure and its Refactored Versions

Java Project	NOCI	LOC	MI	RLC	MCI	CC
Jadventure	60	4925	151.54	2.87926	1.955	1.955
$Jadventure_{LM\text{-}GC\text{-}FE}$	68	5272	151.61	2.72898	1.877	1.877
$Jadventure_{LM\text{-}FE\text{-}GC}$	64	5219	151.61	2.671368	1.869	1.869
$Jadventure_{GC\text{-}LM\text{-}FE}$	64	5220	151.40	2.667668	1.866	1.866
$Jadventure_{GC\text{-}FE\text{-}LM}$	64	5170	151.61	2.683497	1.877	1.877
$Jadventure_{FE\text{-}LM\text{-}GC}$	64	5192	151.61	2.759302	1.871	1.871
$Jadventure_{FE\text{-}GC\text{-}LM}$	62	5145	151.61	2.789624	1.883	1.883

Metric Values for Javaanpr and its Refactored Versions

Java Project	NOCI	LOC	MI	RLC	MCI	CC
Javaanpr	66	4782	150.49	0.0112	1.685488	1.95
$Javaanpr_{LM\text{-}GC\text{-}FE}$	75	5276	150.66	0.0102	1.536732	1.835
$Javaanpr_{LM\text{-}FE\text{-}GC}$	82	5490	150.66	0.0098	1.476468	1.762
$Javaanpr_{GC\text{-}LM\text{-}FE}$	77	5415	150.32	0.0107	1.608424	1.78
$Javaanpr_{GC\text{-}FE\text{-}LM}$	77	5439	150.32	0.0106	1.593392	1.774
$Javaanpr_{FE\text{-}LM\text{-}GC}$	73	5283	150.32	0.0109	1.638488	1.824
$Javaanpr_{FE\text{-}GC\text{-}LM}$	74	5369	150.32	0.0108	1.623456	1.79

Experimental Results (Contd.)

Metric Values for CommonBeansUtils and its Refactored Versions

Java Project	NOCI	LOC	MI	RLC	MCI	CC
CBS	111	11734	148.85	0.0102	1.51827	2.267
$CBS_{LM\text{-}GC\text{-}FE}$	216	14035	150.62	0.0089	1.340518	2.254
$CBS_{LM\text{-}FE\text{-}GC}$	215	14110	151.23	0.0087	1.315701	2.249
$CBS_{GC\text{-}LM\text{-}FE}$	216	14267	150.69	0.0086	1.295934	2.262
$CBS_{GC\text{-}FE\text{-}LM}$	216	14108	151.2	0.0088	1.33056	2.25
$CBS_{FE\text{-}LM\text{-}GC}$	217	14203	150.6	0.0088	1.32528	2.252
$CBS_{FE\text{-}GC\text{-}LM}$	217	14119	150.3	0.0088	1.32264	2.251

Metric Values for Pagseguro and its Refactored Versions

Java Project	NOCI	LOC	MI	RLC	MCI	CC
Pagseguro	132	9075	41.52	0.0036	0.149472	1.756
$Pagseguro_{LM\text{-}GC\text{-}FE}$	142	9452	46.19	0.0032	0.147808	1.706
$Pagseguro_{LM\text{-}FE\text{-}GC}$	143	9464	56.95	0.0022	0.12529	1.698
$Pagseguro_{GC\text{-}LM\text{-}FE}$	141	9444	46.53	0.0030	0.13959	1.706
$Pagseguro_{GC\text{-}FE\text{-}LM}$	141	9405	50.71	0.0032	0.162272	1.713
$Pagseguro_{FE\text{-}LM\text{-}GC}$	142	9447	50.65	0.0033	0.167145	1.708
$Pagseguro_{FE\text{-}GC\text{-}LM}$	141	9403	50.75	0.0030	0.15225	1.712

Experimental Results (Contd.)

Metric Values for GnuBridge and its Refactored Versions

Java Project	NOCI	LOC	MI	RLC	MCI	CC
GBG	160	11362	106.42	0.0068	0.723656	1.486
$GBG_{LM-GC-FE}$	179	12515	111.78	0.0040	0.44712	1.477
$GBG_{LM-FE-GC}$	177	12477	111.81	0.0044	0.491964	1.477
$GBG_{GC-LM-FE}$	179	11490	106.75	0.0045	0.480375	1.48
$GBG_{GC-FE-LM}$	178	12541	115.82	0.0036	0.416952	1.48
$GBG_{FE-LM-GC}$	179	12504	109.85	0.0040	0.4394	1.477
$GBG_{FE-GC-LM}$	179	12537	112.24	0.0043	0.482632	1.479

Metric Values for Dungeon and its Refactored Versions

Java Project	NOCI	LOC	MI	RLC	MCI	CC
Dungeon	197	9988	154.64	0.0041	0.634024	1.69
$Dungeon_{LM-GC-FE}$	283	11567	157.11	0.0045	0.706995	1.686
$Dungeon_{LM-FE-GC}$	282	11786	157.11	0.0044	0.691284	1.686
$Dungeon_{GC-LM-FE}$	283	11259	157.11	0.0047	0.738417	1.686
$Dungeon_{GC-FE-LM}$	281	11871	160.16	0.0048	0.768768	1.665
$Dungeon_{FE-LM-GC}$	284	11487	155.22	0.0047	0.729534	1.656
$Dungeon_{FE-GC-LM}$	283	11562	168.90	0.0050	0.8445	1.656

Experimental Results (Contd.)

Metric Values for FernFlower and its Refactored Versions

Java Project	NOCI	LOC	MI	RLC	MCI	CC
FLR	225	28989	78.46	0.0017	0.133382	3.414
$FLR_{LM-GC-FE}$	252	29217	155.48	0.0023	0.357604	3.356
$FLR_{LM-FE-GC}$	253	29209	155.48	0.0023	0.357604	3.355
$FLR_{GC-LM-FE}$	255	29164	150.78	0.0020	0.30156	3.367
$FLR_{GC-FE-LM}$	254	29160	155.23	0.0019	0.294937	3.369
$FLR_{FE-LM-GC}$	252	29205	154.91	0.0022	0.340802	3.356
$FLR_{FE-GC-LM}$	252	29152	154.91	0.0022	0.340802	3.37

Metric Values for Jsmpp and its Refactored Versions

Java Project	NOCI	LOC	MI	RLC	MCI	CC
Jsmpp	307	18873	68.46	0.0047	0.321762	1.581
$Jsmpp_{LM-GC-FE}$	317	19754	110.57	0.0035	0.386995	1.58
$Jsmpp_{LM-FE-GC}$	319	19759	119.15	0.0035	0.417025	1.569
$Jsmpp_{GC-LM-FE}$	318	19555	118.7	0.0036	0.42732	1.62
$Jsmpp_{GC-FE-LM}$	319	19566	118.4	0.0036	0.42624	1.59
$Jsmpp_{FE-LM-GC}$	319	19455	95.64	0.0039	0.372996	1.57
$Jsmpp_{FE-GC-LM}$	318	19546	96.34	0.0038	0.366092	1.57

Experimental Results (Contd.)

- **Analysis of Results**
  - To achieve maintainable software, the analysis has been done individually on the metrics considered.
- **Analysis of Relative Logical Complexity Values**
  - Out of the 16 projects analysed, 8 projects produced minimum values of RLC when the code smell elimination sequence performed is LM-FE-GC.
  - For other permutations left, from 12.5% to 25% of the projects produce minimum values of RLC.
  - The sequence LM-FE-GC gives the best results if RLC is considered.
- **Analysis of McCabe's Cyclomatic Complexity Values**
  - It is found that 56.25% of the projects under study give minimum value of CC by performing the sequence LM-FE-GC.
  - Sequence GC-FE-LM produces maintainable codes for 6.25% of the projects.
  - FRVs formed by GC-FE-LM have very few unique independent paths while the FRVs generated by LM-FE-GC have a good amount of independent paths.

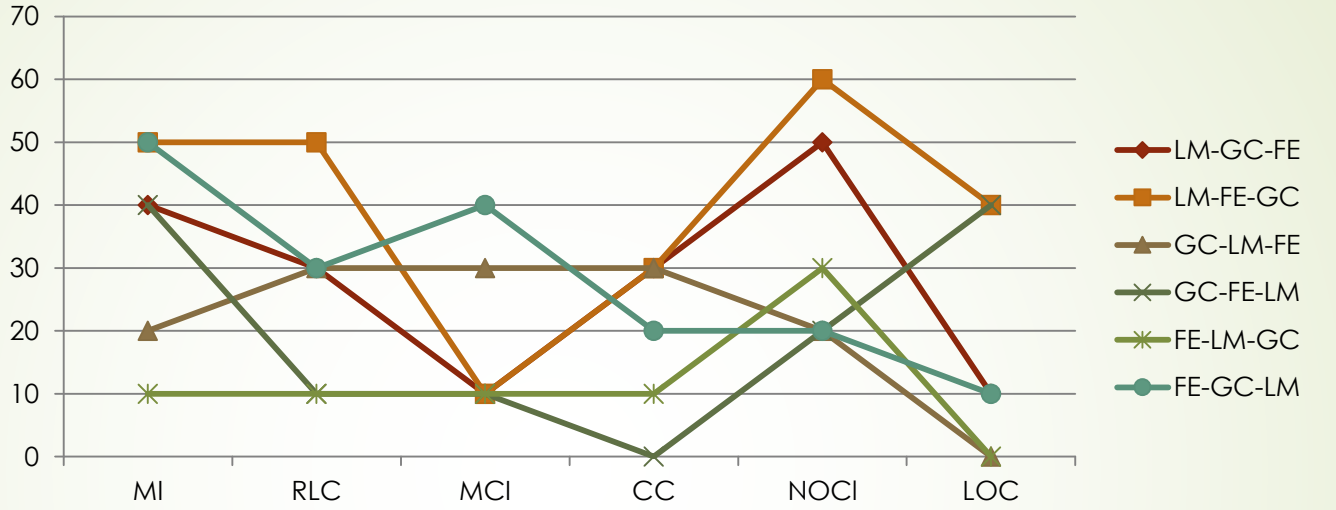
Experimental Results (Contd.)

- **Analysis of Maintainability Index Values**
  - 50% of the total FRVs examined projects have maximum MI if code smells are removed in a sequence of LM-FE-GC.
  - 50% of the total FRVs examined projects have maximum MI if code smells are removed in a sequence of LM-FE-GC.
- **Analysis of Maintainability Complexity Index Values**
  - It is observed that 25% of the refactored versions of the examined systems produce the maximum value of MCI if the sequences LM-FE-GC, GC-LM-FE or FE-GC-LM are followed.
  - 18.75% of the refactored versions produced by following the sequences LM-FE-GC or GC-FE-LM have maximum MCI value.
  - Only 12.5% projects have the highest MCI value if FE-LM-GC sequence is preferred to eliminate the code smells.

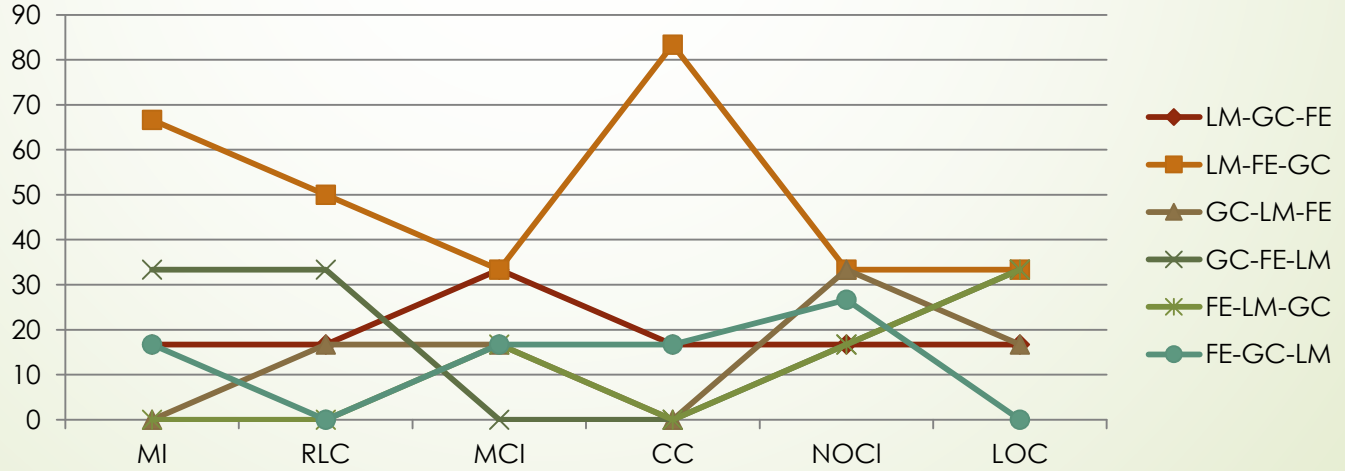
Experimental Results (Contd.)

- **Analysis of NOCI and LOC Values**
  - It is found that 50% of the FRVs have maximum number of classes if LM-FE-GC code smell elimination sequence is followed.
  - 43.75% refactored projects have maximum lines of code by following LM-FE-GC.
- **Analysis of Set I and Set II Projects**
  - Keeping MI in mind, 50% Set I projects and 66.66 % Set II projects have maximum MI if code smells are removed in a sequence of LM-FE-GC.
  - For small projects belonging to Set I, 50% projects have minimum values of RLC for the sequence LM-GC-FE and another 50% Set II projects yielded minimum values of RLC for LM-FE-GC.
  - By considering MCI for Set I projects, it is found that 40% of FRVs have maximum MCI if the sequence FE-GC-LM is followed and 33.33% Set II projects have maximum MCI by following the sequence LM-FE-GC or LM-GC-FE.
  - As far as CC is concerned, LM-GC-FE, LM-FE-GC and GC-LM-FE prove to be the best, while large projects have minimum CC when LM-FE-GC is followed.
  - By comparing the results of NOCI, LM-FE-GC produces maximum reults for both Set I and Set II.
  - FRVs produced by sequence LM-FE-GC have maximum lines of code for both Set I and Set II projects.

Maintainability Trends

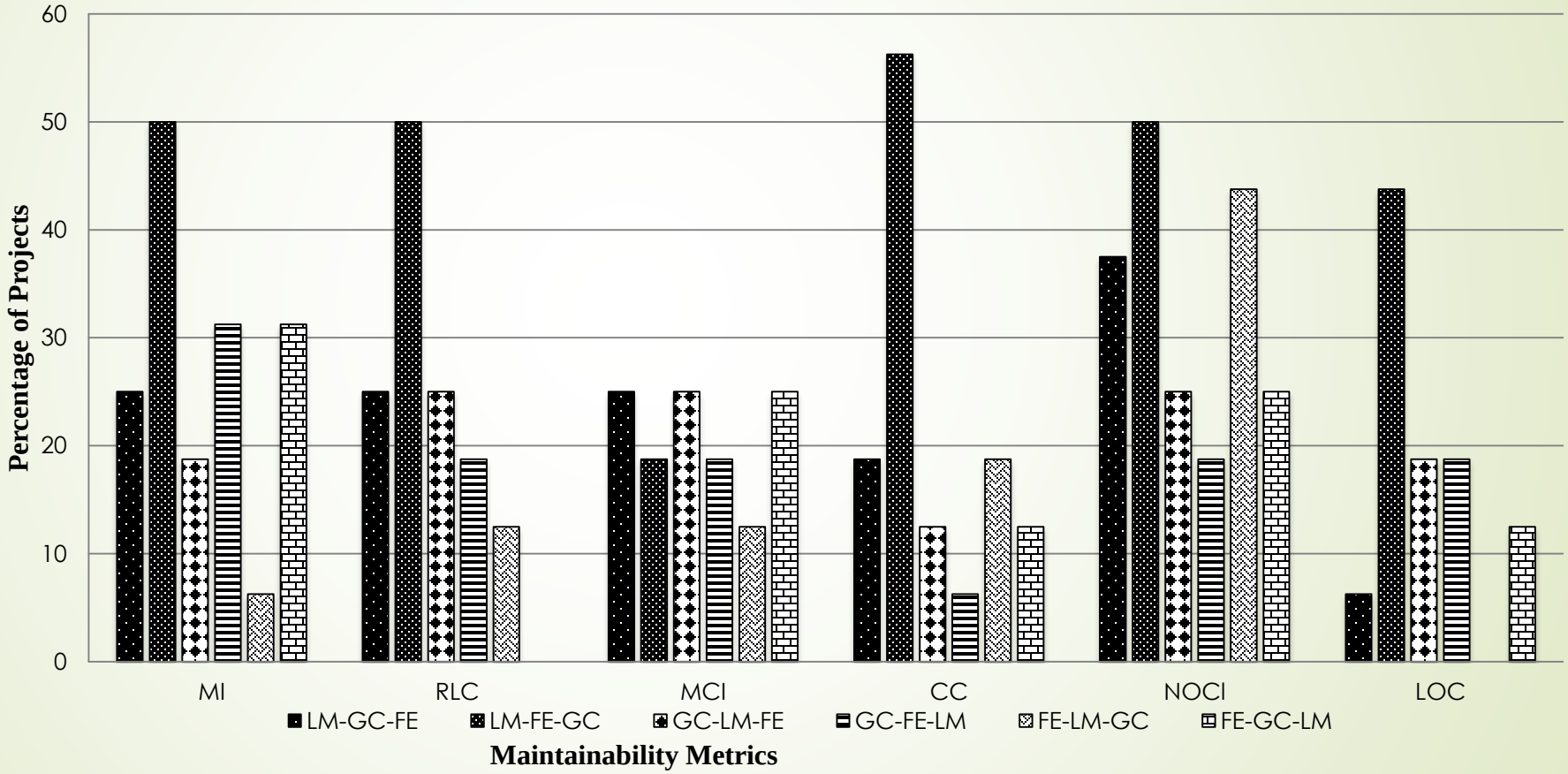


Maintainability Trends across Applications in Set I



Maintainability Trends across Applications in Set II

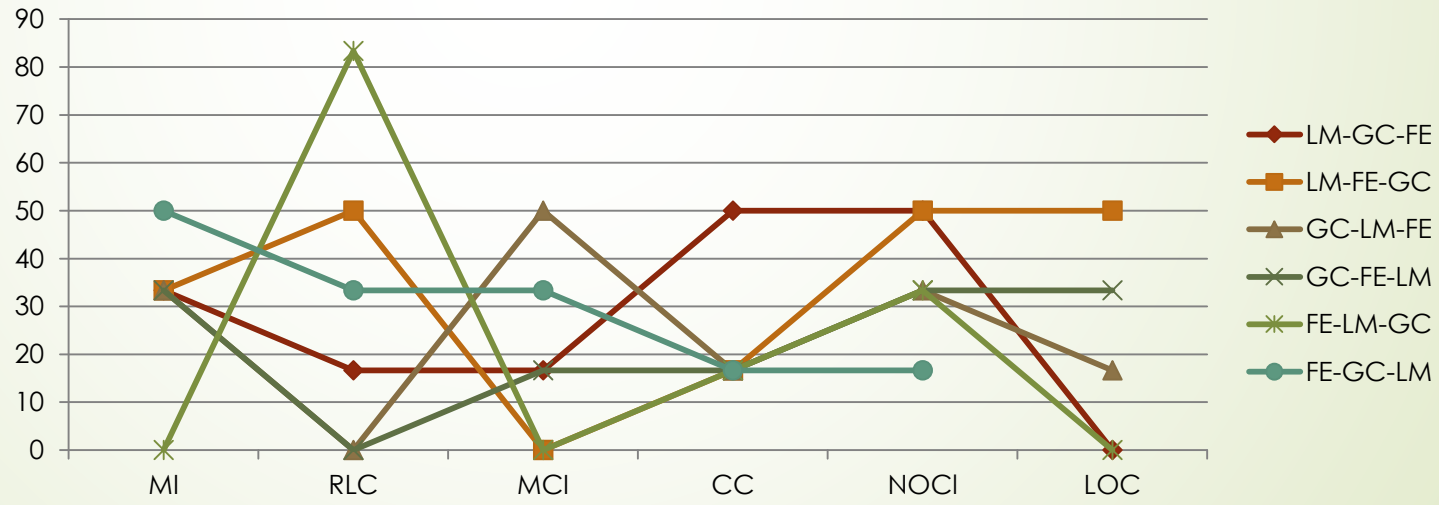
Maintainability Trends (Contd.)



Graphical Depiction of Percentage of Projects against Maintainability Metrics

Analysis of Similar Code Smell Instances Projects

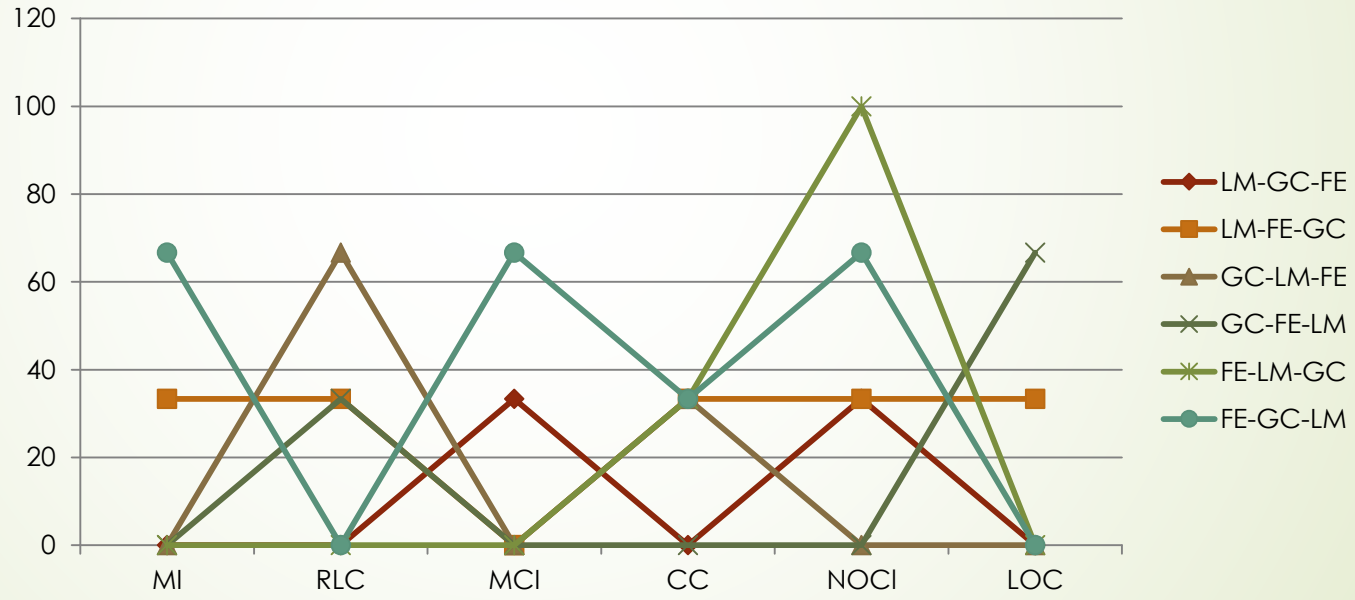
- The subject systems analysed are divided into 4 types in which the projects present under each Type contain similar ratio of code smell instances.
- The Type 1 consists of 6 projects, Spinfo, Frogger, TigerIsland, JavaGame, Jpacman and JavaBoatGame.



Trend Analysis of Applications under Type 1

Analysis of Similar Code Smell Instances Projects (Contd.)

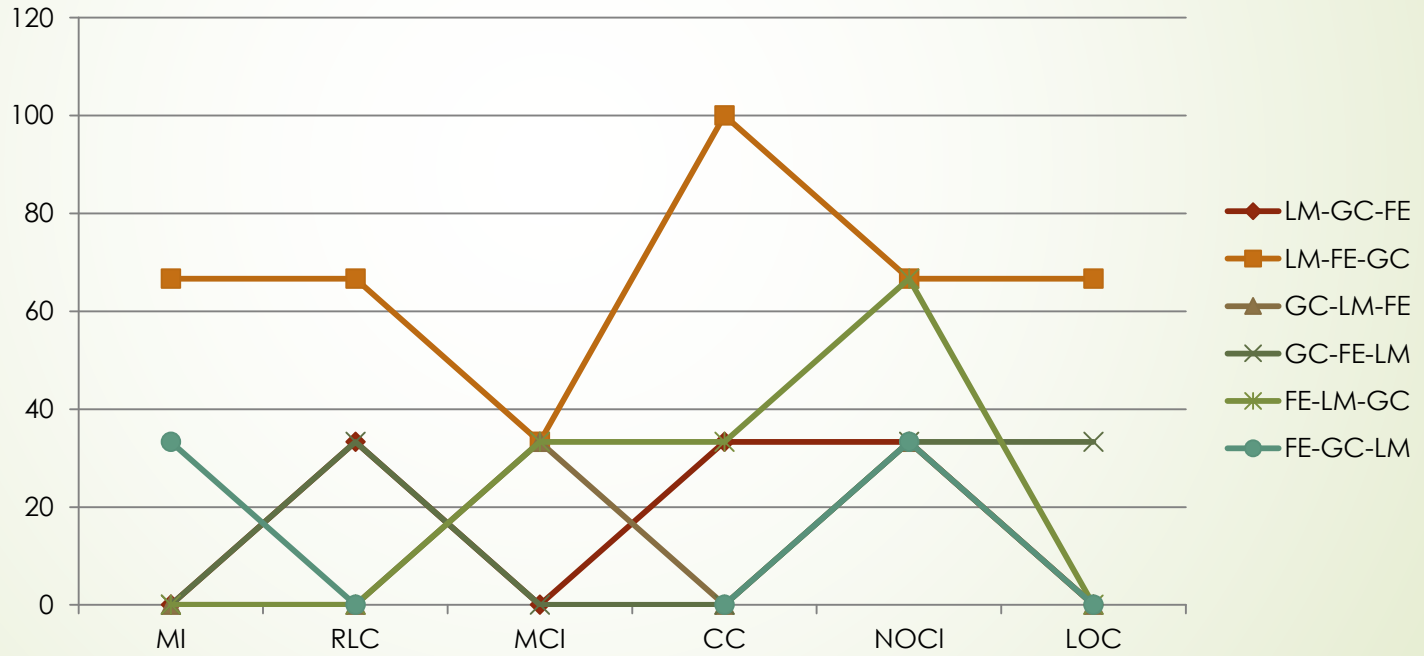
- Type 2 applications include Algorithms, CommonBeansUtils (CBS) and Dungeon.



Trend Analysis of Applications under Type 2

Analysis of Similar Code Smell Instances Projects (Contd.)

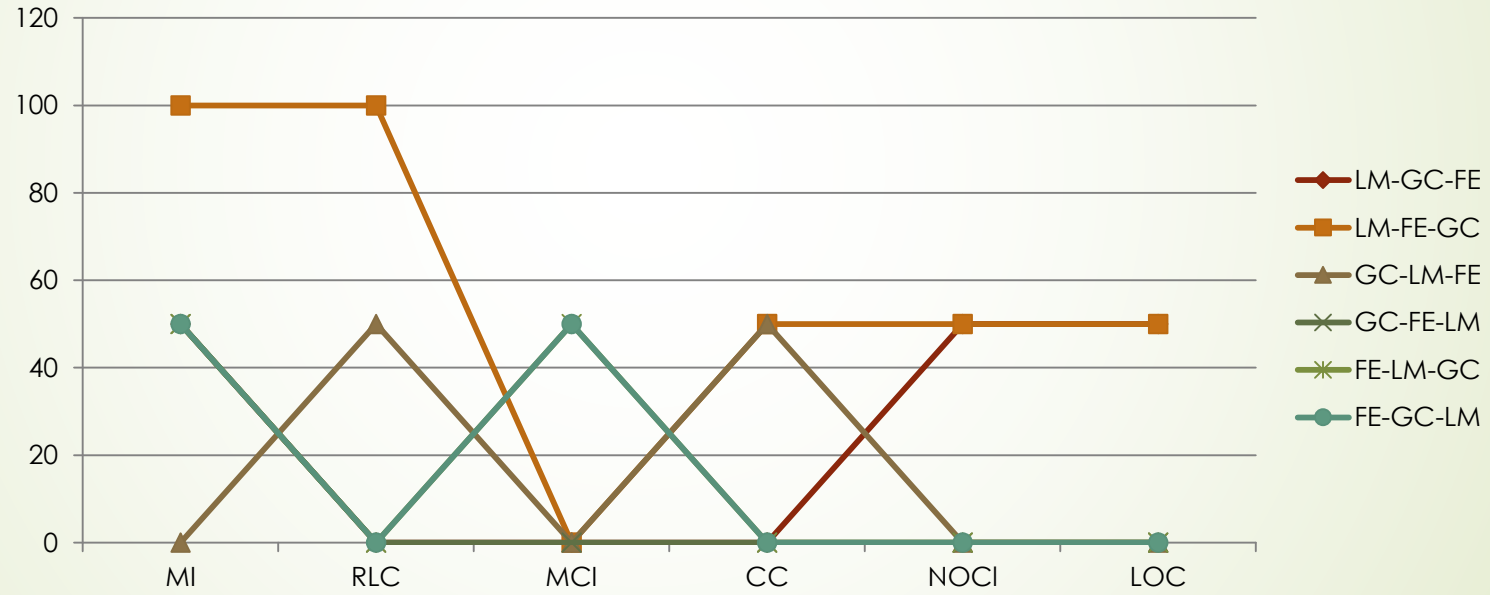
- The Type 3 applications include Jsmpp, Pagseguro and GnuBridge.



Trend Analysis of Applications under Type 3

Analysis of Similar Code Smell Instances Projects (Contd.)

- The Type 4 applications include Jadventure and Javaanpr.



Trend Analysis of Applications under Type 4

Conclusion

- We empirically analysed and identified the code smell elimination sequence to achieve high quality software.
- 6 metrics affecting the quality of the code are chosen namely, Maintainability Index, Relative Logical Complexity, Maintainability Complexity Index, McCabe's Cyclomatic Complexity, Number of Classes and Lines of Code.
- The study is performed on two sets of sample applications, set I containing projects with less than 100 classes and set II consisting of applications with more than 100 classes.
- The study is performed on 16 java applications so we get 16 IRVs.
- The code smell free final application created is named as final refactored version (FRV).
- Thus we get 96 (16 * 6) FRVs. These 96 FRVs thus obtained are analysed for finding the results.

Conclusion (Contd.)

- The results obtained suggest that LM-FE-GC sequence yields the most maintainable and the highest quality software.
- The refactored application contains maximum number of classes and maximum lines of code.

Analysis of Code Smell Removal Sequences against Maintainability

Maintainability Metrics	Code Smell Removal Sequence for Maximum Value	Code Smell Removal Sequence for MinimumValue
MI	LM-FE-GC	FE-LM-GC
RLC	LM-FE-GC	FE-GC-LM
MCI	FE-GC-LM	FE-LM-GC
CC	LM-FE-GC	GC-FE-LM
NOCI	LM-FE-GC	GC-FE-LM
LOC	LM-FE-GC	FE-LM-GC

Future Work

- Impact on other characteristics such as testability and program comprehension can also be observed by code smell removal.
- Only three code smells have been removed but other code smells removal might also be impactful in the maintenance of the software.
- Some other class prioritization technique can also be developed to detect the classes which are in the utmost need of refactoring.
- The relation between maintainability and code smell removal sequence might be different for the applications developed on other platforms such as C++ or Python.

References

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

[2] Sharma, T., Suryanarayana, G. and Samarthyam, G., 2015. "Challenges to and solutions for refactoring adoption: An industrial perspective." *IEEE Software*, *32*(6), pp.44-51.

[3] Marinescu, R., 2004, September. "Detection strategies: Metrics-based rules for detecting design flaws." In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on* (pp. 350-359). IEEE.

[4] Palomba, F., 2015, May. "Textual analysis for code smell detection." In *Proceedings of the 37th International Conference on Software Engineering-Volume 2* (pp. 769-771). IEEE Press.

[5] Wongpiang, R. and Muenchaisri, P., 2013, November. "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* (pp. 160-164). IEEE.

[6] Mansoor, U., Kessentini, M., Bechikh, S. and Deb, K., 2013. "Code-smells detection using good and bad software design examples." *Technical report, Technical Report*.

[7] Rasool, G. and Arshad, Z., 2015. "A review of code smell mining techniques." Journal of Software: Evolution and Process, 27(11), pp.867-895.

[8] Olbrich, S.M., Cruzes, D.S. and Sjøberg, D.I., 2010, September. "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems." In Software Maintenance (ICSM), 2010 IEEE International Conference on (pp. 1-10). IEEE.

References

[9] Ouni, A., Kessentini, M., Bechikh, S. and Sahraoui, H., 2015. "Prioritizing code-smells correction tasks using chemical reaction optimization." Software Quality Journal, 23(2), pp.323-361.

[10] Tarwani, S. and Chug, A., 2016, September. "Sequencing of refactoring techniques by Greedy algorithm for maximizing maintainability." In Advances in Computing, Communications and Informatics (ICACCI), 2016 International Conference on (pp. 1397-1403). IEEE.

[11] Pérez-Castillo, R. and Piattini, M., 2014. "Analyzing the harmful effect of god class refactoring on power consumption." IEEE software, 31(3), pp.48-54.

[12] Anda, B., 2007, October. "Assessing software system maintainability using structural measures and expert assessments." In Software Maintenance, 2007. ICSM 2007. IEEE International Conference on (pp. 204-213). IEEE.

[13] Vidal, S.A., Marcos, C. and Díaz-Pace, J.A., 2016. "An approach to prioritize code smells for refactoring." Automated Software Engineering, 23(3), pp.501-532.

[14] Singh, P., "Minimizing Refactoring Effort through Prioritization of Classes based on Historical, Architectural and Code Smell Information."

[15] Malhotra, R., Chug, A. and Khosla, P., 2015, August. "Prioritization of classes for refactoring: A step towards improvement in software quality." In Proceedings of the Third International Symposium on Women in Computing and Informatics (pp. 228-234). ACM.

[16] Yamashita, A. and Moonen, L., 2012, September. "Do code smells reflect important maintainability aspects?." In Software Maintenance (ICSM), 2012 28th IEEE International Conference on (pp. 306-315). IEEE.

References

[17] Peters, R. and Zaidman, A., 2012, March. "Evaluating the lifespan of code smells using software repository mining." In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on* (pp. 411-416). IEEE.

[18] Travassos, G., Shull, F., Fredericks, M. and Basili, V.R., 1999, October. "Detecting defects in object-oriented designs: using reading techniques to increase software quality." In *ACM Sigplan Notices* (Vol. 34, No. 10, pp. 47-56). ACM.

[19] Dhaka, G. and Singh, P., 2016, December. "An Empirical Investigation into Code Smell Elimination Sequences for Energy Efficient Software." In *Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific* (pp. 349-352). IEEE.

[20] Tsantalis, N., Chaikalis, T. and Chatzigeorgiou, A., 2008, April. "JDeodorant: Identification and removal of type-checking bad smells." In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on* (pp. 329-331). IEEE.

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

[22] Frappier, M., Matwin, S. and Mili, A., 1994. "Software metrics for predicting maintainability." *Software Metrics Study: Tech. Memo*, *2*.

[23] Meananeatra, P., 2012, September. "Identifying refactoring sequences for improving software maintainability." In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (pp. 406-409). ACM.