

Copyright is owned by the Author of the thesis. Permission is given for a copy to be downloaded by an individual for the purpose of research and private study only. The thesis may not be reproduced elsewhere without the permission of the Author.

ON THE AUTOMATION OF DEPENDENCY-BREAKING REFACTORINGS IN JAVA

A THESIS PRESENTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN
COMPUTER SCIENCE
AT MASSEY UNIVERSITY, PALMERSTON NORTH,
NEW ZEALAND.

SYED MUHAMMAD ALI SHAH

2013

Abstract

Over a period of time software systems grow large and become complex due to un-systematic changes that create a high level of interconnection among software artefacts. Consequently, maintenance becomes expensive and even making small changes may require considerable resources due to change propagation in the system, a phenomenon known as *ripple effects*. Industrial evidence suggests that more resources are spent on the maintenance phase than on the initial development. It is evident that companies make huge investments to maintain legacy systems until a point comes where a complete restructuring of the system is required. In most cases, it becomes very expensive to refurbish legacy systems manually due to their inherent complexity. Several semi-automated solutions have been proposed to restructure simplified models of existing systems. It is still expensive, in terms of resources, to translate those model level transformations into source code transformations or refactorings. The question that arises here is whether we can automate the application of model level changes on the source code of programs.

In this thesis, we have developed novel algorithms to automate the application of a class of architectural transformations related to improving modularity of existing programs. In order to evaluate our approach, we have analysed a large dataset of open source programs to determine whether the manipulation of models can be translated into source code refactorings, whether we can define constraints on those refactorings to preserve program correctness, and to which extent the automation of the whole process is possible. The results indicate that this automation process can be achieved to a significant level, which implies that certain economic benefits can be gained from the process.

Acknowledgements

I would like to express my profound gratitude to my supervisor Dr. Jens Dietrich for his advice, guidance, and endless support through every step of the way. He has a great passion for his work and he knows how to get the best out of his students. He has been a source of learning throughout these years. I also thank him for the Guery tool he had developed. This tool was very helpful in my research.

I extend my sincere gratitude to my co-supervisor Dr. Catherine McCartin for her valuable feedback during the entire time. She has been very kind and helped me polish my work. This work would have not been possible without the help and support of my supervisors.

A special thanks to my family. Words cannot express how grateful I am to my mother (late), and father for all of the sacrifices that they have made on my behalf. Their prayer for me was what sustained me thus far. Thanks to all of my siblings for their support and wishes during these years. I would like to thank my sisters Farhat, Yasmin, and Najma for looking after me so well, whenever I visited home. A word of thanks to my younger brother Taskeen, who helped me in many different ways during my research.

My years at Massey were very enjoyable, thanks for the friendship of Abrar (for all the enjoyable distractions), Shujat (for the jokes and laughs), Ezanee (for stimulating useful discussions), Saleem (for being a good flatmate and friend), and Tariq (for the friendship and discussions on every aspect of life). I would also like to thank my friends in Pakistan especially Irfan, Bhatti, Ainan, and Naseer. Many thanks to the Pakistani community in New Zealand. They have been very kind and never made me miss the exotic Pakistani cuisine. They all made me feel at home.

I take this opportunity to thank Massey University and the School of Engineering and Advanced Technology (SEAT) for providing a conducive working environment for research. I would also like to thank the SEAT staff, in particular, Christina Bond, Fiona, Michelle Wagner, Linda Lowe, and Dilantha Punchihewa for never complicating the administrative tasks.

Last but not least, I would like to thank the Higher Education Commission for providing the financial support granted through Overseas Scholarship Program to complete my PhD degree.

This thesis is dedicated to my late mum ...

Contents

Abstract	iii
Acknowledgements	v
1 Introduction	1
1.1 Problem Definition	1
1.2 Research Questions	7
1.3 Approach	8
1.3.1 Critical Dependency Detection	8
1.3.2 Tools	9
1.3.3 The Dataset	9
1.4 Thesis Contribution	10
1.4.1 Algorithms	10
1.4.2 Implementation	10
1.4.3 Validation	10
1.5 Thesis Structure and Outline	11
2 Research Methodology	13
2.1 Architectural Model - The Dependency Graph	13
2.1.1 Extracting the Model	14
2.2 Architectural Antipatterns	14

2.2.1	Antipattern Detection Tools	15
2.2.2	Evaluation of Tools	15
2.2.3	Representing Antipatterns	17
2.3	Antipattern Set	19
2.3.1	Overview	19
2.3.2	Circular Dependencies between Packages	20
2.3.3	Subtype Knowledge	23
2.3.4	Abstraction Without Decoupling	26
2.3.5	Degenerated Inheritance	28
2.4	Detecting Opportunities - Scoring Edges	31
2.5	Dependency Classification	33
2.6	The Dataset	34
3	Dependency-Breaking Refactorings	37
3.1	Overview	37
3.2	Package Level Refactorings	39
3.2.1	Move Class	39
3.2.2	Split Packages	40
3.2.3	Merge Packages	41
3.3	Class Level Refactorings	42
3.3.1	Adapt Parameter	42
3.3.2	Extract Interface	45
3.3.3	Dependency Injection	46
3.3.4	Service Locator	50
3.3.5	Type Generalisation	51
3.3.6	Static Members Inlining	52
3.4	Evaluation of Refactorings	56

4	Applying Package Level Refactorings	61
4.1	Overview	61
4.2	Background	63
4.3	Algorithm	64
4.3.1	Building the Dependency Graph	66
4.3.2	Computing Antipattern Instances	66
4.3.3	Computing Edge Scoring	67
4.4	Implementation: CARE - The Eclipse Plugin	67
4.4.1	Implementing Dependency Classification	68
4.4.2	Implementing Refactoring Constraints	68
4.4.3	Implementing Refactorings	72
4.5	Strongly Connected Component Metrics Definition	75
4.6	Experiment	78
4.6.1	Case Study: JMoney-0.4.4	78
4.6.2	Case Study: JGraph-5.13.0	79
4.6.3	Impact of Move Class Refactoring	81
4.6.4	Refactoring Simulation on Model vs Refactoring Application on Code	82
4.6.5	Impact of Program Size on Number of Refactorings	83
4.6.6	Package Merging	84
4.6.7	Distribution of Move Refactorings	85
4.6.8	Refactorability	85
4.6.9	Success Estimation of Model to Code Refactorings	86
4.6.10	Strongly Connected Components Metrics	87
4.6.11	Limitations of the Experiment	88
4.6.12	Scalability	93
4.6.13	Test Results	93

4.7	Summary	95
5	Applying Composite Refactorings	97
5.1	Overview	97
5.2	Background	98
5.2.1	Type Generalisation	98
5.2.2	Service Locators	100
5.2.3	Static Members Inlining	100
5.3	Algorithm	101
5.3.1	The Dependency Graph	102
5.3.2	Computing Antipattern Instances	103
5.3.3	Computing Edge Scoring	103
5.3.4	Parsing Source Code	103
5.4	Implementation: CARE - The Eclipse Plugin	103
5.4.1	Implementing Dependency Classification	103
5.4.2	Implementing Refactoring Constraints	105
5.4.3	Implementing Refactorings	108
5.5	Experiment	112
5.5.1	Examples	112
5.5.2	Impact of Refactorings on Instance Count Metric	114
5.5.3	Refactoring Simulation on Model vs Refactoring Application on Code	114
5.5.4	Refactoring Types Applied	115
5.5.5	Strongly Connected Components Metrics	115
5.5.6	Test Results	117
5.6	Summary	117
6	Conclusions and Future Work	123

6.1	Research Questions	123
6.1.1	Can model level dependency-breaking refactorings be automatically translated into source code refactorings?	124
6.1.2	How can we define and evaluate constraints on refactorings to preserve the correctness of the program being refactored?	124
6.1.3	To what extent can these dependency-breaking refactorings be automated?	125
6.2	Threats to Validity	126
6.2.1	Dataset Selection	126
6.2.2	Correctness of Refactored Programs	126
6.2.3	Developers Feedback	127
6.2.4	Influence of Tools	128
6.2.5	Java Specific Refactorings	128
6.2.6	Scalability	129
6.3	Research Contributions	129
6.4	Future Work	130
	Bibliography	131
	A Declaration of Previous Work	141
	B CARE Plugin: Installation and Instructions	143
B.1	Installation	143
B.1.1	Configuration	143
B.2	Usage Instructions	144
B.2.1	User Interface	144
B.2.2	Preferences	144
B.2.3	Importing Projects	144
B.2.4	Refactoring Output	145

List of Tables

2.1	Tool Features in terms of Architectural Antipatterns Detection	16
2.2	Comparison of Different Scoring Mechanisms	32
2.3	The Dataset	36
3.1	Refactoring Attributes in terms of Breaking Dependencies	58
4.1	Instance Count Before and After Refactoring	72
4.2	Metric Values of Three SCCs	77
4.3	The Resultant Move Refactorings for JMoney-0.4.4	78
4.4	Metrics Values for JMoney-0.4.4	80
4.5	Metrics Values for JGraph-5.13.0	80
4.6	Result for Merged Packages	85
4.7	Refactorability Example	86
4.8	Top 5 Programs with Highest Execution Time	94
4.9	Test Results of 5 Programs Before and After Refactorings	95
5.1	Dependency Categories and their Default Respective Refactorings	105
5.2	Test Results of 5 Programs Before and After Refactorings	117
B.1	Eclipse Project Structure	145
B.2	Dataset Files	145
B.3	Output Description	146

List of Figures

1.1	Evolution of Packages and Classes in JRE	4
1.2	Evolution of Package Relationships and Class Relationships in JRE	4
1.3	Evolution of Package and Class level Tangles in JRE	5
1.4	Evolution of Relationships of Package and Class level Tangles in JRE	5
1.5	JRE 1.7.0 Package Level Dependency Graph	6
1.6	JRE 1.7.0 Class Level Dependency Graph	6
2.1	User Interface/Database Dependency Antipattern	17
2.2	Dependency Cycle Between the AWT and Swing Packages	22
2.3	Circular Dependency between Packages	22
2.4	Weak Circular Dependency between Packages	22
2.5	Abstraction Example	23
2.6	Example of Subtype Knowledge Antipattern	24
2.7	Subtype Knowledge	26
2.8	Example of Abstraction Without Decoupling Antipattern	27
2.9	Abstraction Without Decoupling	29
2.10	Example of Degenerated Inheritance Antipattern	30
2.11	Degenerated Inheritance	31
2.12	Example Program's Dependency Graph	32
3.1	Move Class Example	40

3.2	Move Class Example	41
3.3	Split Packages Example	42
3.4	The Adapt Parameter Refactoring	44
3.5	Extract Interface Refactoring	46
3.6	Example of Dependency Injection	48
4.1	Class Diagram of Dependency Classification for Move Refactoring	68
4.2	Class Diagram of Pre and Postconditions	69
4.3	Example of Increase in the Instance Count Metric	73
4.4	Example of Decrease in the Instance Count Metric	74
4.5	Class Diagram of Refactorings	75
4.7	Package Dependency Graph of JMoney-0.4.4	79
4.8	Decrease in SCD Instances After Move Refactorings	81
4.9	Decrease in no. of Instances: Comparison between SCD and WCD	82
4.10	Decrease in no. of Instances: Comparison on Model and Code Levels	83
4.11	Impact of Program Size on Number of Refactorings	84
4.12	Refactorability	87
5.1	Automated Refactoring Process	102
5.2	Class Diagram of Dependency Categories	104
5.3	Class Diagram of Composite Refactorings	109
5.4	Decrease in Instance Count Metric After Refactorings	115
5.5	Decrease in no. of Instances: Comparison on Model (graph) and Code levels	116
5.6	Refactoring Types Applied	117
B.1	CARE Installation	147
B.3	CARE Preferences	149
B.4	Import Existing Projects	150

B.5 Select Projects to Import 151