Master’s Thesis

A Reverse Software Engineering Process to Ascertain Architecture Conformance through Conceptual Re-modularisation and Imposing Architecture Paradigm - Experience with Open Source Code

Ahmad Elghafari

March 13, 2012

Department of Information Systems
Graduate School of Information Science
Nara Institute of Science and Technology
A Master’s Thesis
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfilment of the requirements for the degree of
MASTER of ENGINEERING

Ahmad Elghafari

Thesis Committee:
Professor Hajimu Iida (Supervisor)
Professor Yoshiki Kinoshita (Co-supervisor)
Professor Hiroyuki Seki (Co-supervisor)
Assistant Professor Norihiro Yoshida (Co-supervisor)
A Reverse Software Engineering Process to Ascertaining Architecture Conformance through Conceptual Re-modularisation and Imposing Architecture Paradigm
- Experience with Open Source Code *

Ahmad Elghafari

Abstract

The literature is rich of reverse software engineering processes and techniques resulting in relative, necessary information to ascertain the software architecture confirms to certain target, intended or reference architecture. There are still various opportunities for engineering researchers to contribute to address one or several requirements of such analysis. To collect the built architecture to be ready for such analyses, a reverse software engineering process is really required. That is to abstract away from lower level of abstraction to higher levels, gradually.

In order to overcome these issues, this thesis introduces a process that re-modularise the built architecture conceptually and imposes architecture paradigm as reference guidelines. As computer and architect collaborate, to capture relative information from built architecture while maintaining vertical abstraction traceability.

The experiment findings suggest that the introduced process is able to ascertain software built architecture carefully. The process shows its applicability to re-modularise difficult built architecture which has enormous cyclic intercommunication patterns. Also, the findings suggest that not all cyclic intercommunication patterns contribute in dissolving the modularity of built architecture.

The proposed process can play as guidance for architects during the early phase of software evolution process. The process inherits the weaknesses of the integrated tools, the vagueness of the referenced architecture paradigm description, the impreciseness of the classification and impact of the various intercommunication patterns, and also the architect expertise and competency level. Future research should investigate whether these weaknesses may be overcome with additional, precise knowledge of architecture paradigm guidelines, deeper analysis and reasoning about the various intercommunication patterns of software system and with the employment of more tools to minimise objectivity.

Keywords:

Acknowledgements

I praise Allah the Creator, the Sustainer and the Evolver of the worlds, Lord of the heavens and of the earth and all that is between them, and Lord of the sun’s risings, who has taught by the pen and has taught man that which be knew not, and who have adorned the lowest heaven with an ornament, the planets. I praise Him, the exalted Majestic, for guiding me to seek Science in Japan and giving me the power and the help to be patient perseverance to continuing my research journey. I am sure He, the Mighty and Majestic, has chosen the best for me to let me experience the research and study of master’s programme in Japan and to be able to realise some wisdoms. And without His infinite support and Mercy I would have not been able to be here today.

I sincerely appreciate the great guidance, support and patience of my supervisor Professor Hajimu Iida, who exerted every single effort to lead me to have my research in its appropriate shape and tried myriad ways to make this stubborn man whose writing this document walks on the right path to the success. Professor Iida has pulled me out, many times, when I almost got drowned in the ocean of Literature, and was seriously concerned about my readings. I was such a very difficult researcher but Professor Iida did not want me to lose my research opportunity. Thank you Professor Iida for everything, I am grateful to you and I appreciate your constructive research guidance taking my hands to let me reach to the finale point. I really enjoyed researching with you, especially the last chapter of my research mission, and without your wisdom and immeasurable diligence I would have not composed a complete thesis.

I would like also to show my appreciation and gratefulness to the co-supervisors of this thesis, Professor Yoshiki Kinoshita and Professor Hiroyuki Seki and Assistant Professor Norihiro Yoshida, for their rich, careful guidance, support and time.

It is the honour of mine to having several research meetings with Professor Yoshiki Kinoshita, the Principal Research Scientist and the Director of Laboratory for Verification and Semantics at the National Institute of Advanced Industrial Science and Technology (AIST), Hyogo Site, Japan. Professor Kinoshita provided me special guidance in the level of software system specifications and design and observation in the dynamic aspect. He welcomed me when I consulted him after one lecture of the course “Recursive Definitions and Inductive Definitions” during the fourth quarter of year 2010. Then I visited him in his office at AIST to discussing my research. Professor Kinoshita dedicated me his time and provided me significant guidance, as I was seeking to bring about some formula as the way I think of the software system. During the meetings, we discussed systematic procedures in system specifications, design and reuse concepts, especially in Automata and Process Algebra concepts. The discussion also reached to notes in Coalgebra and Timed Automata, and Dependable Systems Engineering. Thank you Professor Kinoshita for all of your guidance, support and time you dedicated me.

I am honoured to having the opportunity to discuss my research with Professor Hiroyuki Seki, the Professor of the Laboratory for Applied Algorithmics. Professor Seki welcomed me when I visited him at his office to discuss my research many times, and dedicated me plenty of his time and special care. Professor Seki showed his interest in my research and provided me significant support every time I visited him. He lent me two books in Theory of Computational Learning of Language Grammar and Automata. I also enjoyed
his personable, clear lectures of the course: “Theory of Computation III” during the third quarter of year 2010. Thank you Professor Seki for all of your support, diligence and time you dedicated me.

I am honoured to be awarded the Japanese Government, Monbukagakusho: The Ministry of Education, Culture, Sports, Science and Technology (MEXT) Research Scholarship 2009, at the Embassy of Japan in Beirut, Lebanon. I would like to thank Mr. Daisuke Okabayashi and Mrs. Gretta Noun for their ultimate support, guidance and advices during the application and screening procedure. Thanks to the Japanese Government, and all the Community who cooperate for the work of this grant. And here I wish to express my gratitude to Professor Chady Elmoucary, the full-time Assistant Professor in the Department of Electrical, Computer and Communication Engineering at the Notre Dame University Louaize, North Lebanon Campus, who recommended me strongly to the Embassy of Japan. I wish also to express my gratitude to Ms. Mona Elmourad, the project manager of the industry I worked at during 2008, who also recommended me strongly to the Embassy of Japan.

I am also honoured to be a recipient of the NEC C&C Foundation Grant to present my paper: “A Proactive Dynamic Reverse Engineering Approach” at the third international conference on Computer Engineering and Technology (ICCET) 2011, Kuala Lumpur, Malaysia. Thanks to NEC C&C personnel for all their sincere effort to support. And once again, thank you professor Iida for strongly recommending me to the organisation.

I am grateful to Professor Yuichi Kaji, the Assistant Professor at the Laboratory for Applied Algorithmics, beginning in his special teaching style and personable, smooth and clearly addressing lectures. He made the complex concept of Shannon entropy easy to understand, as I attended his course: “Information Theory” during the first quarter of year 2010. Professor Kaji gave the lectures in Japanese language, however, he annotated the slides with English language notes. Professor Kaji also showed me his interest in my research and that was a boost to my morale, and did not hesitate to give me his support whenever I visited him in his office or contacted him by email. Thank you Professor Kaji for all of your support, time and consideration.

At the Software Design and Analysis laboratory, I would like to thank the Assistant Professor Norihiro Yoshida and the Assistant Professor Kyohei Fushida for their ultimate support and valuable comments and feedback and literature suggestions. And a special thank to Ms. Yuuki Ro for her dedicated care of all the laboratory’s office business. I would also like to thank all of the colleagues at the laboratory, especially Mr. Kenji Fujiwara, Mr. Raula Kula and Mr. Yang Xin and Mr. Rikichi Hirayama and Mr. Masaki Obana and Mr. Shunsuke Yoshioka and Mr. Yuta Yamada and Mr. Shingo Ishimura and a very special thank to Mr. Kazuki Hamasaki. Thanks for all of your support during my research time at the laboratory, especially for the sincere comments and feedback you provided me about my presentation slides in the last week. Mr. Hamasaki provided me more comments when I was enhancing my slides and told me where he feels my slides are difficult for potential audience.

I wish to show sincere appreciation to all academic and management members of Nara Institute of Science and Technology (NAIST), Graduate School of Information Science, for
their countless efforts and deviations to supporting students’ research career. Thanks to all the professors whom with I attended classes or met in some seminar or workshop, especially Professor David Sell and professor Mike Barker for their personable lectures. And special thanks and appreciation to the Student Affair Division personnel, who provide countless support on the minute bases, for their extreme efforts, especially Ms. Ayako Ohta and Mr. Norito Hamada.

I appreciate the generosity of Ambassador of Lebanon in Tokyo, Japan and all his support. Special thanks to him and to Ms. Sayuri Harada for her guidance and promptness.

Special thanks to all friends and colleagues, whom I spent my months of two years and a half at NAIST, for their valuable advices and support, especially Mr. Mauricio Alexandre Parente Burdelis and Mr. Kato Takashi and Mr. Asad Habib and Mr. Doudou Fall and Mr. Gregory Blanc and Ms. Michiko Matoba and Mr. Sami Alfattani and Mr. Rafael Torres and Ms. Ratna Aisuwarya (Ana) and Ms. Chittaphone Phonarath (Nooi) and Mr. Marcos Villagra (Danny) and Mr. Devisetti Naveen and Ms. Mwathi Jane Wamaitha and Ms. Elga Strafella. Thanks also to the colleagues who organise for the Friday Prayer, and thanks once again to NAIST for the support.

Thanks to all colleagues in Osaka for their support during my stay there and yet after. Myriad thanks to Ms. Mai Hoa for the friendly atmosphere you always make. And, a very special thank to Mr. Simón Enrique Ortiz Branco, the first scholar I met when I first arrived to Japan in April 2009, for helping me during the early days in Osaka. A special thank also to Doctor Rafeef Alzoubi, Faculty of Engineering, Osaka University, for her ultimate sincere diligence, support and uncountable help.

I also would like to thank the teachers of the International Student Centre at Osaka University where I attended the 4-month intensive Japanese language programme from April to July 2009, especially the director Professor Takako Muraoka and Professor Tomoko Smith.

During my research career in Japan I met many colleagues with whom either shared knowledge and experience of scientific research or casual conversations. I would like to thank Professor Daniel Collado-Ruiz, Associate Professor at Universidad Politécnica de Valencia Spain, for telling his experience in attending conferences and discussing Engineering Process Designing concept and relative topics. My thanks also to Ali bin Aqeel Khan for discussing various topics in Organisation Management and relative topics.

I would like also to conclude my appreciation to all colleagues and friends and acquaintances who were there as some part in the evolution of building the success of this research either with direct or indirect ways and affects, as I dedicate you my poem, “Friendships are Like Dews From Clouds”, that I have written during my stay in Japan. And thanks once again for all your encouragements.

What gracious moments!
Just in the freeze,
on the island of geyser breeze.
Moments aroused,
bringing gifts from unexpected worlds;
All that silence for all the loudness brought.
Loud, overloud as ocean and rocks blast;
Loud, overloud,
entertain does the soul.
How wonderful!
Friendships are like dews from clouds;
The dews bringing new life
carrying highly the soul
as gathering petals of a rose;
Grayness to greenness evolves.

My pure reverence and gratitude to my sister and brother and mother and father who are always there and after Allah they are the main resource of morale and encouragement, as they invested and sacrificed their lives to help me to begin and accomplish my research challenge, and without their devotion and patience with me to get my former degree I would have not come to Japan.

And peace be unto the apostles, the Prophets and Messengers! And praise be to Allah, the Lord and Cherisher of the Worlds!
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td></td>
</tr>
<tr>
<td>1.1 State of the Practice</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Software Evolution Process</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Software Architecture</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Reverse Software Engineering</td>
<td>5</td>
</tr>
<tr>
<td>1.5 Software Architecture Conformance Analysis</td>
<td>6</td>
</tr>
<tr>
<td>1.6 Problem Statement</td>
<td>7</td>
</tr>
<tr>
<td>1.7 Proposed Process Overview</td>
<td>8</td>
</tr>
<tr>
<td>1.8 Contribution</td>
<td>8</td>
</tr>
<tr>
<td>1.9 Thesis Outline</td>
<td>10</td>
</tr>
<tr>
<td><strong>2 Previous Published Techniques</strong></td>
<td></td>
</tr>
<tr>
<td>2.1 Techniques for Entire Process</td>
<td>11</td>
</tr>
<tr>
<td>2.1.1 Require Development Activities</td>
<td>11</td>
</tr>
<tr>
<td>2.1.2 Lack Traceability across Vertical Abstraction Levels</td>
<td>12</td>
</tr>
<tr>
<td>2.1.3 Miss Conformance to Architecture Paradigm</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Specialised Techniques</td>
<td>12</td>
</tr>
<tr>
<td>2.2.1 Regulate Architecture Constraints into Generic Ones</td>
<td>12</td>
</tr>
<tr>
<td>2.2.2 Analyse without Sufficient, Careful Reasoning</td>
<td>13</td>
</tr>
<tr>
<td><strong>3 Proposed Process</strong></td>
<td></td>
</tr>
<tr>
<td>3.1 The Recovery of Built Architecture</td>
<td>17</td>
</tr>
<tr>
<td>3.1.1 Architecture Modularity and Cyclic Dependency</td>
<td>17</td>
</tr>
<tr>
<td>3.2 Conceptual Re-modularisation</td>
<td>19</td>
</tr>
<tr>
<td>3.2.1 Class Level Re-modularisation Algorithm</td>
<td>20</td>
</tr>
<tr>
<td>3.2.2 Package Level Re-modularisation Rules</td>
<td>21</td>
</tr>
<tr>
<td>3.3 Imposing Reference Architecture Paradigm</td>
<td>22</td>
</tr>
<tr>
<td>3.3.1 Defining a Logical Architecture</td>
<td>22</td>
</tr>
<tr>
<td>3.3.2 Implementing Architecture Paradigm Guidelines</td>
<td>22</td>
</tr>
<tr>
<td>3.4 Ascertainment of the Built Architecture</td>
<td>23</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Reverse and Forward Software Engineering Processes</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>Software Architecture Recovery Process</td>
<td>6</td>
</tr>
<tr>
<td>1.3</td>
<td>The Proposed Process Overview</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Java File and Class Level Cyclic Dependency</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>The Cyclic Dependency in The Package Level</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>Distinguishable Levels of Software Source Code Package</td>
<td>18</td>
</tr>
<tr>
<td>3.4</td>
<td>Logical Categories from Packages and Classes</td>
<td>21</td>
</tr>
<tr>
<td>3.5</td>
<td>The Ascertainment of the Built Architecture Process Overview</td>
<td>24</td>
</tr>
<tr>
<td>3.6</td>
<td>MVC Architecture Paradigm Model</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>The Manual Recovery of JHotDraw Framework Package</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>The Manual Recovery of JHotDraw Standard Package</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>The Logical Categories of JHotDraw Framework Package</td>
<td>32</td>
</tr>
<tr>
<td>4.4</td>
<td>The Logical Categories of JHotDraw Standard Package</td>
<td>33</td>
</tr>
<tr>
<td>4.5</td>
<td>Initial Logical Categories of JHotDraw Standard Package</td>
<td>33</td>
</tr>
<tr>
<td>4.6</td>
<td>The Final Categories of JHotDraw Framework and Standard Packages</td>
<td>34</td>
</tr>
<tr>
<td>4.7</td>
<td>Partial Categories of JHotDraw with the Assigned Classes</td>
<td>35</td>
</tr>
<tr>
<td>4.8</td>
<td>The Logical Architecture Top Abstract Level for JHotDraw</td>
<td>35</td>
</tr>
<tr>
<td>4.9</td>
<td>The Logical Layers of JHotDraw with the Categories</td>
<td>37</td>
</tr>
<tr>
<td>4.10</td>
<td>The Logical Architecture of JHotDraw</td>
<td>38</td>
</tr>
<tr>
<td>4.11</td>
<td>Cyclic Dependency between DrawingView and Handle in JHotDraw</td>
<td>40</td>
</tr>
<tr>
<td>4.12</td>
<td>The Model Layer Dependency on View Layer in JHotDraw</td>
<td>40</td>
</tr>
<tr>
<td>4.13</td>
<td>The View Layer Dependency on Model Layer in JHotDraw</td>
<td>41</td>
</tr>
<tr>
<td>4.14</td>
<td>The View Layer Dependency on Controller Layuer in JHotDraw</td>
<td>41</td>
</tr>
<tr>
<td>4.15</td>
<td>The Controller Layer Dependency on View Layer in JHotDraw</td>
<td>42</td>
</tr>
<tr>
<td>4.16</td>
<td>The Controller Layer Dependency on Model Layer in JHotDraw</td>
<td>43</td>
</tr>
<tr>
<td>4.17</td>
<td>Sonargraph Analysis and MVC Paradigm</td>
<td>44</td>
</tr>
<tr>
<td>4.18</td>
<td>The Internal Structure of Gui Package in JEdit</td>
<td>46</td>
</tr>
<tr>
<td>4.19</td>
<td>The Logical Categories of JEdit Gui Package</td>
<td>46</td>
</tr>
<tr>
<td>4.20</td>
<td>The Logical Categories of JEdit Bsh Package</td>
<td>46</td>
</tr>
<tr>
<td>4.21</td>
<td>The Logical MVC Architecture Defined for JEdit</td>
<td>51</td>
</tr>
<tr>
<td>4.22</td>
<td>Model Layer of JEdit with the Assigned Logical Categories</td>
<td>52</td>
</tr>
<tr>
<td>4.23</td>
<td>The Final Logical Architecture Defined for JEdit</td>
<td>52</td>
</tr>
<tr>
<td>4.24</td>
<td>Cyclic Dependencies in Package Category Level</td>
<td>58</td>
</tr>
<tr>
<td>4.25</td>
<td>JEdit Source code Statistics in Sonargraph Dashboard</td>
<td>59</td>
</tr>
<tr>
<td>4.26</td>
<td>Partial Role Modelling View</td>
<td>62</td>
</tr>
</tbody>
</table>
4.27 Partial OOGs View .............................................. 63
4.28 All Dependencies of Handle Interface in JHotDraw ................. 64

A.1 Logical Category ................................................. 74
A.2 Sub Layer ......................................................... 74
A.3 Layer ............................................................ 74
A.4 Package .......................................................... 75
A.5 Class .............................................................. 75
A.6 Interface .......................................................... 76
A.7 Green Outgoing Dependency ...................................... 76
A.8 Green Ingoing Dependency ...................................... 76
A.9 Red Outgoing Dependency ...................................... 76
A.10 Red Ingoing Dependency ....................................... 76
# List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Intercommunications characteristics of Object Oriented Classes</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>Subject Software Systems Source Code General Statistics</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>The Architecture Recovery Effort Spent on JHotDraw 5.1</td>
<td>30</td>
</tr>
<tr>
<td>4.3</td>
<td>Assigned Categories to each MVC Layer in JHotDraw</td>
<td>36</td>
</tr>
<tr>
<td>4.4</td>
<td>The Dependency among JHotDraw Layers in DSM Table</td>
<td>39</td>
</tr>
<tr>
<td>4.5</td>
<td>The Logical Categories Assigned to MVC Layers for JEdit - Part 1</td>
<td>48</td>
</tr>
<tr>
<td>4.6</td>
<td>The Logical Categories Assigned to MVC Layers for JEdit - Part 2</td>
<td>49</td>
</tr>
<tr>
<td>4.7</td>
<td>The Dependency among JEdit MVC Layers in DSM Table</td>
<td>51</td>
</tr>
<tr>
<td>4.8</td>
<td>Dependencies characteristics of View Layer on Controller Layer in JEdit</td>
<td>53</td>
</tr>
<tr>
<td>4.9</td>
<td>Dependencies characteristics of View Layer on Model Layer in JEdit</td>
<td>54</td>
</tr>
<tr>
<td>4.10</td>
<td>Dependencies characteristics of Model Layer on View Layer in JEdit</td>
<td>55</td>
</tr>
<tr>
<td>4.11</td>
<td>Dependencies characteristics of Controller Layer on Model Layer in JEdit</td>
<td>55</td>
</tr>
<tr>
<td>4.12</td>
<td>Dependencies characteristics of Controller Layer on View Layer in JEdit</td>
<td>56</td>
</tr>
<tr>
<td>4.13</td>
<td>Cyclic Dependencies in Class and Packages Levels in JEdit</td>
<td>58</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The craft of software engineering has crossed the frontier of software industry through medical institution, aerospace and automotive industry, and so forth through the development of embedded systems. Nowadays, myriad of scientific applications and medical devices, especially automotive innovations which are solely based on complex software systems with 90 percent are driven by software. For instance, the S-Class of Mercedes-Benz contains more than 600,000 lines of code [22]. Software systems account for 40 percent of a vehicle’s cost [3]. The responsibility is big on software engineers. A reliability of these large, complex software systems in the real world must be guaranteed or human loss is a penalty — not only a fatal error appears on a PC screen. Indeed, the advancement in software engineering is a potential and boost to the reliability and rigorousness of the other technology lines products.

During 1960s, software engineers realised the need for special treatments in developing large scale software systems [44]. Software architecture was born and grew to have its own descriptive notations, and to integrate with the software development process models.

Software architecture is an essential abstraction artefact during the development of large software, especially object oriented software systems. Therefore, the expertise of software architects affects the quality of the in-vehicle software architecture or in any other software-intensive innovations that play major roles among people’s daily life.

Software engineers found themselves once again in front of huge complex code that need to be represented in a reasonable way to fit human cognitive capabilities and show required information. In 1980s, software engineers was intuitively inspired by the hardware reverse engineering concept and adapted it to have models and automated tools that can analyse large software systems — reverse software engineering [13].

Today, in large scale software crafting, the question is much harder: “how can one make sure that the built architecture conforms to the target architecture?” And one more question: “how can one make sure that the target architecture conforms to universally accepted architecture guidelines?”
The work presented in this thesis aims to provide a process that exploits reverse software engineering capabilities to ascertain the software system built architecture. The mechanism based on re-modularising conceptually and hierarchically recovered built architecture and imposing an architecture paradigm.

This chapter is organised as follows. Section 1.1 presents the state of the software engineering craft problem, the importance of software system in humans’ lives and the challenges. Section 1.2 introduces software evolution process as natural demand that large object oriented software systems exhibit. Section 1.3 discusses the importance of software architecture in engineering large software systems. Section 1.4 talks about reverse software engineering as a major abstracting technique for certain analyses and reasoning. Section 1.5 presents briefly the role of software architecture conformance analysis and architecture recovery. Then in Section 1.6, I present the problem statement. I give an overview of the proposed process in Section 1.7. I outline the contribution of this thesis in Section 1.8. Finally, Section 1.9 concludes this chapter with an outline for the rest of this document.

1.1 State of the Practice

In many technology lines products, include telecommunication, medical or mechanical vehicles, software plays a dominant role today [7]. “The average fortune-100 company has 35 million lines of code in operation with a growth of 10 percent per year” [10].

One big difference between vehicle or any hardware manufacturing and software development is that software is more likely to change and evolve. This is a great chance dedicated by the nature of software for engineers crafting systems − whether purely software or hybrid of a software system with mechanic or others. It permits engineers to react against the risk that software systems become outdated and unsatisfying to the stakeholders, through continuous amendments to their software systems which is the software evolution process.

Unfortunately, this opportunity comes with extra layers of difficulties. The software evolution process requires careful reasoning about the observed architecture and behaviour of subject software systems.

Corporations accept the challenge of software evolution. A similar challenge exists with the open source software movement. Both corporation software and open source software evolve over several years, and some have existed since decades like UNIX. These systems are large and complex, and in particular the object oriented software, embody substantial knowledge, including requirements, design decisions [10], and architecture artefacts. Keep in mind that, and usually, design documents in industry do not exist or they are outdated [51]. It was my surprise when I asked about the design documents during my first week of my former industrial experience. After reading some articles, I realised it is a famous, typical story. It happens also that open source projects do not provide design documentations [12]. Under this assumption the knowledge about subject software is difficult to recover after many years of evolution and personnel change [10]. Fortunately, the source code still exists it is the remedy and the pain at the same time. How? The truth lies in the code lines makes itself the only reliable software artefacts for comprehension [25, 11]. In contrast, a software
engineer cannot grasp the architecture of complex, large source code which may span over hundreds of thousands pages.

The second challenge is to prepare the software system source code for the conformance analysis. That is to recover its architecture structure, to abstract it in higher levels and to reasonably present the relative information.

The third challenge is to make sure that the recovered structure of the software making its architecture adheres to the architecture guidelines and best practice, and this is the output of the architecture conformance analysis.

1.2 Software Evolution Process

Software obeys to the “law of continuing change” as one law discovered by Lehman and Belady in [31]. That is the software evolution is the process of progressive change of software systems [36]. It is the set of activities, both technical and managerial, that ensures that software continues to meet organisational and business objectives in a cost effective way [35]. The evolution process of a corporation or open source community is in their hands. They may turn it to be effective and result in better quality new software releases or it will be the degradation and the end of the software project life.

Developers often may not be familiar with the original architecture [45]; especially being new comers, or no sufficient management or software project sharing knowledge mechanism. Moreover, software development team members may lose the track, especially when being pressed with deliverables deadlines. They tend to fix defects locally [27], or rely on ad hoc, code statement resolutions for problems, e.g., bugs, within subsystems or particular build units. Therefore, without looking to the global structure of the system and the interdependencies among the other subsystems and build units this work-around fix may, and most probably, violate architecture design decisions and rules. Some developers feel the pleasure of maintaining changes and fixing bugs with trivial source code tricks, and repeating this kind of changes the system will get increasingly less comprehensible and more complex going through its architecture erosion era!

Moreover, developers may find it useful to rely on subsystems of the software to fulfil feature’s functionality, therefore forming dependencies on the subsystems that are not permitted by the original architecture of software. It is a trick of software: the source code cannot constraint any kind of amendment to any of its lines. And this is how the evolution of a software system may become very costly and work expensive [50] and a reason behind architecture erosion and drift [44], “Software erosion is the decreasing quality of the internal structure of a software system” [44, 50].

Those are examples of the law discovered by Lehmans and Belady “ignorant surgery” [31]. These kinds of changes make software systems less reusable, hard for unit testing, and less reliable [30] and more rigid [37].

Obviously, conducting a software evolution process needs a technology and treatments that
can handle large scale software systems. The technology should result in source code reengineering activities while maintaining and carefully dealing with the whole structure of the system and the original design objectives. Modifications should not be directly made to the source code. Instead, it should be through design decisions to the software architecture which can be absorbed and translated to source code lines.

This is the role of software architecture which I discuss in the following section.

1.3 Software Architecture

As motivated above, large scale object oriented software systems contain huge amount of constituent elements and their intercommunication patterns building the structure of the system architecture. Therefore, the engineering and evolution of such kind of systems require the notion of software architecture which is defined as “the fundamental organisation of a system, embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution” [15].

It is possible to grasp the information about a small system all at once, however, it is impossible of large systems [44, 45]. Software architecture may not be seen from writing small programs [20, 12]. However, practitioners share their experiences with the researchers through international conferences and workshops. Researchers are aware of the large and complex real world software projects. They consider industrial crises as motivations which drive their works to provide resolution methods and drawing theories for the evolving discipline – the software architecture and its analyses. In addition, the open source movement resulted in realistic software systems having large quantities of code [12]. This allows researchers to realise the importance of software architecture and conduct researches in its area.

Software architecture allows engineers to look at the system from different viewpoints to describe it in its entirety [28, 45, 48]. The architecture definitely resides in the source code, however it is not explicit [17]. Therefore, source code cannot permit to view different perspectives or obtain different levels of abstraction or even just bring to the audience only relative information for certain task or need.

Moreover, software architecture has built mechanism and language for transformations through different views and levels of abstraction in both vertical and horizontal dimensions. Indeed most of us are interested in the vertical dimension, especially when they go in gradual pace from the lower levels to the higher levels. However, as many things that are not easy with software, the abstraction skills is not an easy capabilities to gain or apply.

In conclusion, software architecture facilitates understanding and analyses of large software systems. Therefore, comprehending the architecture of such systems is crucial, especially in the early phase of software evolution process [6]. As in different engineering disciplines, such as Civil Engineering, the architecture is what the contractor, project manager, architecture engineers, assistants, quality impact analysts, a third party quality assurance personnel and many others gather around an architecture blueprint of some concrete building project and
discuss, evaluate and conclude. Likewise, the large software architecture has the difficulty compared to skyscrapers and needs a medium that can bring about for its stakeholders reasoning and analysis capabilities. We name it Architecture.

1.4 Reverse Software Engineering

The complexity of large software system architecture makes understanding such kind of systems impossible for software engineers. A large software system is not easily understandable [11]. Therefore, the early phase of software evolution process, before the actual amendments, requires more than 50 percent of the time of the whole process [27]. Software engineers need to have sufficient understanding of the system to start their software evolutions processes. This is where reverse software engineering comes into the play. One purpose of conducting reverse engineering is to recover software architecture [45]. This discipline had been absorbed from hardware engineering [13] developing methodologies and tools to support program understanding and handle software architecture extraction, abstraction and representation. That is it is the process that analyses the subject system’s artefacts and extracts the necessary information to aid the practitioners in the comprehension [11, 13], and gain a sufficient understanding of the system [45], in the architecture or design level. It is to analyse a subject system and to recover any unavailable documentation, and then, visualise the recovered information through raising the abstraction to a higher level [45]. A forward software engineering process goes from certain specifications and intended, or target architecture toward building the system. The target architecture is implemented with system source code which makes the built architecture. Therefore, two main differences between reverse and forward software engineering, is that the reversing process (1) goes from low abstract levels to higher or different abstract levels (vertically or horizontally) and (2) does not attempt any amendments to the system built architecture. Figure 1.1 illustrates the relation of the Reverse and Forward software engineering processes. Usually,

![Figure 1.1: The Reverse and Forward Software Engineering Processes.](image)

in industry design documents either do not exist or they are outdated [51]. Open source software projects do not often come with documents or explicit specifications of functional or Quality of Service (QoS) requirements [20]. Assuming some design documents reflect the current implementation of source code (i.e., the built architecture), the manual investigation still does not pay off well. Therefore, the starting point of a reverse engineering process is the system itself [13] as it is most often the only available and reliable artefact which is the
source code or from different perspective, the built architecture [25, 11, 45].

1.5 Software Architecture Conformance Analysis

As motivated above, the high abstract level architecture and design documentation of a software system is often not available and impossible to be grasped by manually investigating the source code. Automating the software architecture recovery is a key technology for having the built architecture of large systems, through a reverse engineering activity. Figure 1.2 illustrates an architecture recovery process.

Software architecture recovery tools with the automated reverse engineering capabilities parse and visualise huge data of the physical, or built, architecture of subject software systems. However, a built architecture is represented in flat view or views that have lots of details and are not customisable which may not match expectations of particular needs, leaving the architects with a new layer of complexity. An architect looking at such kind of views would say: “And then what?” The worse is when showing the class level view of a package, especially a fat one, contains more than 500 to 2000 top level class. And once again, software architects cannot handle the comparison of both architecture models or mediums of a large system.

The tools get deceived because of the architecture under the analysis which originally does not maintain modularity among its building artefacts. From source code point of view, the classes and packages are probably involved in some cyclic intercommunication patterns that breach the modularity of their structuring. One consequence is that the hierarchy among such elements get dissolved and the modularity of the software architecture faint. And therefore, the architecture recovery process thinks that the whole system is just one layer; or it is a box of toys thrown; or even it is a bunch of wool yarns have been missed up by a cat. Moreover, in any possible way to visualising the built architecture, the original intention of the architects and the global architecture of the top level abstract cannot be captured from the built architecture.

Around 18 years ago visualisation was defined as “the process of transforming information into a visual form, enabling users to observe the information. The resulting visual display enables the scientist or engineer to perceive visually features which are hidden in the data but nevertheless are needed for data exploration and analysis” [21]. So the traditional
convention about visualisation was that the information about artefacts of a system is visualised by the architecture recovery process and the interpretation and abstraction is left to the human user. This is not a challenge for architecture but a shortage in the level of results and abstraction that an automated process can bring about. Automated techniques should be more usable than that. On the other hand, some practitioners believe that a representation technique should filter relevant from irrelevant information [46] and give practitioners only the information they ask for each time. That is practitioners do not usually ask for complete information about a subject system [46]. Therefore, the key usefulness of architecture recovery is in how reasonable, scalable, and efficient the extracted information is visualised.

In object oriented software systems, it may be easy to detect classes and packages and to observe the physical structure in a coarse-grained level. But it is a hard problem to reason about the interdependencies among classes and packages, especially for a large scale object oriented software system. Class and package level of abstraction is not enough to comprehend the architecture of software system [5]. Moreover, object oriented programming paradigm has a reputation of making systems harder to understand [55]. And such interdependencies may contribute in architecture violation. Analysing all these interdependencies by manual investigation of plain recovered architecture is infeasible. Moreover, leaving the analysis to architects means relying on subject, biased ascertainment results. Architects may have different insights about a software architecture artefact, and also experience plays a major role in the analysis that each architect can suggest.

Therefore, practitioners and scholars have contributed with lots of architecture conformance analysis processes and techniques which (1) provide transformation and abstraction mechanisms to abstract plain views of recovered architecture away to higher levels, (2) manage recovered architecture, (3) synchronise system implementation with target architecture and (4) address well defined architecture guidelines and ascertain whether built architecture adheres to target architecture and discover where it exhibits violations and how much it deviates and degrades.

In conclusion, an architecture analysis process should aid in identifying the relevant, necessary information for checking the compliance of the architecture with certain target architecture or architecture paradigm guidelines [5].

1.6 Problem Statement

The purpose of software architecture conformance analysis is to ascertain whether built software architecture conforms to reference architecture.

With these motivations and factors in mind, the thesis argues that the problem of architecture conformance analysis can be addressed through exploiting conceptual re-modularisation and imposing reference architecture paradigm, that should meet the following demands:

1. Provide guidance to ascertain the quality of built software architecture.
2. Reason carefully about instances of architecture violations in the built architecture.

1.7 Proposed Process Overview

The two key solutions are (1) bringing back the modularity of software system by conceptually re-modularising its building elements, and (2) imposing certain architecture paradigm to be used as a reference. The re-modularisation should hierarchically rearrange the elements of built architecture and abstract them into higher abstract levels gradually. Then, a logical architecture for the built architecture is defined when a chosen reference architecture paradigm is imposed. The conceptual re-modularisation is an abstraction mechanism which provides traceability across abstracts of different levels. The imposition of architecture paradigm provides a chance to conduct architecture conformance analysis that results in careful reasoning about the built architecture quality. It also locates where the built architecture obeys to the architecture rules and how it violates. The conformance analysis is done through differentiating the built architecture against the chosen architecture paradigm imposed. The proposed process consists of four main processes as follows.

1. Recovery of built architecture.
2. Conceptual re-modularisation of the built architecture.
3. Imposition of reference architecture paradigm through logical architecture.
4. Ascertainment of built architecture.

The output results of the approach are the final logical architecture implementing the reference architecture paradigm guidelines with a set of descriptions of where the built architecture violates the reference paradigm guidelines and reengineering suggestions. Figure 1.3 illustrates an overview of the proposed process and Chapter 3 provides the description.

1.8 Contribution

The contributions of my thesis are three folds:

1. Architecture conformance analysis with respect to a reference architecture paradigm. I introduce a guidance of how to conduct conformance analysis against architecture paradigm exploited as reference criteria.
2. Feasibility of approach. The feasibility is shown in experimental settings.
3. Comparison to previous studies. I compare my results with two previous techniques.

In summary, The results of the two case studies suggest that the introduced process is able to perform several reverse software engineering phases in vertical abstract dimension and to
Figure 1.3: Overview of the Proposed Reverse Software Engineering Process to Ascertain Architecture Conformance.
ascertain software built architecture carefully. The process in this way guides the architect to realise the next higher level of abstraction each time leading to the top level architecture. The process shows its applicability to re-modularise a difficult built architecture which has enormous cyclic intercommunication patterns. Also, the findings suggest that not all cyclic intercommunication patterns contribute in dissolving the modularity of built architecture. I discuss my experiment results in Chapter 4 and I discuss further my findings in Chapter 5.

1.9 Thesis Outline

This thesis is organised as follows.

In the following Chapter 2, I review related work in software architecture conformance analysis. I explain the proposed process in Chapter 3, all along with definition of terminologies used and discussion about each process of the proposed approach. In Chapter 4, I evaluate the process in unfolding experimental settings with two case studies. I start with describing the necessary tools. Then, I describe the subject systems. After that, each case study for a subject system is discussed. The findings are discussed at the end of each case study. I proceed to discuss the threats to validity of the results. Finally, I demonstrate the contribution of my process by comparison with two related studies. In Chapter 5, I discuss the contributions and limitations of the thesis. I also sketch opportunities and draw potential research lines for future. This chapter also concludes the thesis.
Chapter 2

Previous Published Techniques

In this chapter, I discuss my review to the literature of software architecture conformance analysis. I organised this chapter in two main sections. The first section introduces the techniques that support an architecture conformance process entirely, whereas the second section talks about some specialised techniques that contribute to some part of the architecture conformance analysis. Each subsection is titled by one of the motivations introduced in Chapter 1, Section 6.

2.1 Techniques for Entire Process

2.1.1 Require Development Activities

I. Code Annotation

Abi Antoun worked on the object oriented software to extract potential runtime objects and represent them in hierarchical views through Ownership Object Graphs (OOGs) [1]. OOGs maintain traceability among abstracts with hierarchical views which address the deficiency of UML class diagrams. Although his tool, SCHOLIA, generates OOGs which are reasonable abstract models and can lead conformance analysis to better result, the tool recovers the built architecture only after a manual code annotation activity. Annotating the code requires some knowledge about the subject software. Therefore, the cost of the architecture recovery is high since the process relies on human involvement. And, in my understanding reverse engineering process should not start with a development activity.

II. Language Extension

Aldrich and colleagues address the problem by transforming a source code written in a programming language to an extended language called ArchJava [2], which enforces architectural concepts seamlessly. This is rather more costly than source code annotation.
That is it requires reengineering activities even before commencing the reverse engineering process. This method requires basic knowledge about the subject system whereas in practice this assumption does not always hold. Especially, for new team comers who have no previous knowledge about the industry’s software project.

2.1.2 Lack Traceability across Vertical Abstraction Levels

Brühlmann and colleagues contributed with a technique based on code annotations, but not similar to Abi Antoun’s technique. The annotation activity is tool supported and their main goal was to capture implicit human knowledge about subject systems [8]. The approach wants not to only use the source code as input data for the analysis, but also considers some knowledge that may hold valuable semantic information about the system which is not explicitly defined in the system’s implementation syntax. This is indeed applicable and of great contribution to architecture analysis. Their study resulted in capturing architecture violation in JEdit architecture; however, they did not provide fine grained reasoning about the violation instances, and they did not link views between the high level and lower levels.

2.1.3 Miss Conformance to Architecture Paradigm

Kuhn and colleagues introduced a semantic clustering approach [29]. The published study facilitates checking built architecture against target one. In their case study on HotDraw, they did not check the recovered architecture how far it complies with the MVC architecture paradigm implemented.

Abi Antoun’s method validates built architecture against target architecture but does not check compliance to architecture paradigm constraints. When I examined JHotDraw OOGs in Abi Antoun’s PhD dissertation, found many interdependencies between objects in the OOGs violate MVC architecture paradigm guidelines. In the comparison activity with Abi Antoun’s study results I elaborate this discussion further in Chapter 4.

The question these studies answer is: does the built architecture conform to the target architecture? However there is also a question about the conformance of the built architecture to the guidelines suggested by the architecture paradigm implemented.

2.2 Specialised Techniques

2.2.1 Regulate Architecture Constraints into Generic Ones

Regulating all possible architecture constraints into a few generic ones inhibit distinguishing all various intercommunication patterns and strengths among constituent elements of software system. Object oriented paradigm provides various kinds of interdependencies which may be established by classes and packages of software system. Such kinds of inter-
dependencies differ in weight or strength, e.g. access, inherit, return, interface call, and so forth.

Sangal and colleagues distinguish between acceptable and unacceptable interdependencies by enforcing design rules to the built architecture[49]. They implemented their analysis in a tool called LDM providing hierarchical representations using Dependency Structure Matrix (DSM) which scales well. But they did not consider the cyclic dependencies among classes. Moreover, LDM allows to define only two types of design rules: can-use and cannot-use. Hence, LDM does not cover all various kinds of intercommunication patterns that may happen within object oriented built architecture. Object oriented paradigm distinguishes different kinds of dependencies, including, use, call and inherit and extend, implement, field type, and so forth. LDM also cannot enforce all possible variations of dependencies suggested by an architecture paradigm. For instance, MVC architecture paradigm permits a dependency from Model layer to View layer which is less intense from its counterpart, i.e. the dependency from View to Model; it is not forbidden totally, but with some consideration.

Some limitations of LDM are addressed by Terra and colleagues with their dependency constraint language (DCL) [51]. DCL allows specifying different types of concrete dependency relations, but still the set of relations supported is not complete.

### 2.2.2 Analyse without Sufficient, Careful Reasoning

There are methods that consider particular intercommunication patterns among software constituent elements by calculating metrics values and highlight any of these patterns that violate architecture constraints. In particular, object oriented intercommunication patterns can be cyclic dependencies which are instances of some kind of collaborations of software source code constituent elements, such as classes and packages. Cyclic dependencies may contribute in violation of the architectural level.

A significant interest has been invested for detecting cyclic dependencies in object oriented software systems. Many tools detect cyclic dependencies and either enumerate them in tables or view them in isolation. However, architects still need to see how these dependencies exist in higher abstract levels of the architecture. Also they want to make sure whether particular cyclic dependencies are really unnecessary, how they exist in the built architecture, and whether they really do contribute in architecture violation.

Most of the cyclic dependency analysis techniques evaluate software systems by generalising all the cyclic dependencies as they contribute in the architecture violation. Many tools calculate the Strongly Connected Component (SCC) or the ratio of packages that involve in cyclic dependencies without fine-grained analysis and special treatments for many exceptional cases. Recently, a great intension have been invested for finer-grained analysis and reasoning about the cyclic dependencies [19, 38, 49].

PASTA is a tool to detect where the source code violates the Acyclic Dependency Principle (ADP)[23]. The approach deals with the cyclic dependency in the package level and ignores it in the class level. Moreover, the approach considers every cyclic dependency is an undesirable one. In many occasions this is not the case. This generalisation can be only applied
to packages do not share a relative role, for example, Core and UI packages.

A unique tool in this area is JooJ which pro-actively detects a potential cyclic dependency between classes as the developer writes the code [39]. The approach computes the Strongly Connected Components (SCC) to detect a potential cycle may exist with a code statement being typed. However, JooJ does not treat the cyclic dependency in the package level. JooJ is not aware with the global architecture of the system code. Therefore, it could be annoying if a developer want to intentionally introduce a cyclic dependency because of some architecture pattern guidelines.

JDepend detects cyclic dependencies among system packages through applying some of Martine’s coupling measures[14]. It does not provide visual representations but only enumerated list in xml schema. That is the mechanism of tracing low level results to higher levels architectural abstracts is not supported.

The approach of Falleri and colleagues focuses on the cyclic dependency in the package level [19]. Their algorithm shows finer-grained analysis than the traditional Strongly Connected Components (SCC) with a metric to prioritise the cyclic dependencies in terms of desirability. They evaluate the level of undesirability of a cycle by a novel metric called diameter. The diameter metric is the computation of the distance between packages involved in the cycle. The diameter computation method is intuitive and it tracks the containment of packages through naming convention and a substantial treatment to distinguish between relatively cohesive packages or not related packages. The diameter metric shows it is applicability to check the cycles resulted because of the lack of MVC architecture paradigm guidelines, however the study still needs to make a linkage to higher levels of abstraction of the built architecture.

All in all, most of these tools do not link architecture level constraints with the detected cyclic dependencies. Moreover, they do not show whether such cyclic dependencies are required because of some architectural guidelines, or because of the requirement to the fulfilment of a system functionality that is necessary and at the same time is not harmful. For instance, in MVC architecture paradigm, Model and View layers have a bi-directional communication pattern. View layer sends a state enquiry and Model layer replies with a change notification. However, the dependency from View to Model is not equal to other way around dependency from Model to View. Dependency detection tools do not take in consideration such sensitive architectural constraints. The tools generalise the harmfulness of cyclic dependencies. Some cyclic dependencies may not be necessarily a symptom of architecture erosion or instances of architecture violation. Careful reasoning and showing how they exist in the built architecture may answer whether the built architecture conforms to architecture paradigm guidelines or not.
Chapter 3

Proposed Process

In this chapter I describe the proposed process in details. First, I give an overview of its phases. Then, I explain each phase in a dedicated section.

As introduced in the first chapter the technique approach is composed of four main processes as illustrated in Figure 1.3. First it recovers the built architecture of subject system, then conceptually re-modularise the built architecture, and then imposes a reference architecture paradigm through defining logical architecture, and finally results in a set of descriptions by validating the built architecture against the reference architecture paradigm.

Each phase of the process should result in a deliverable shown in Figure 1.3. These intermediate output will be used together in the final process making the final output, which is reasoning about built architecture how it conforms to well defined guidelines and where it breaches the rules.

The first process recovers the built architecture of the subject system. Here, I would like to clarify for you the three different kinds of architecture names the thesis uses, the built, target and logical architectures.

The built architecture of a software system is the observed architecture when parsing or reading the physical structure of the source code of the subject system. The source code is the main input for the first phase. The source code must be compileable and executable code written in Java programming language. The recovery process parses the (java) with (.class) or (.jar) files to analyse the subject system and provides the visualisations of the built architecture. This step is fully tool supported using Sonargraph. I introduce Sonargraph in Chapter 4.

Software system, in the context of this thesis, is an object oriented system and defined as: a set of packages and their dependency relations. A package is a set of classes and their dependency relations. Dependency among packages and classes can be acyclic or cyclic, and direct or transitive. Packages and classes can have cyclic dependency when they depend on
each other at the same time. I discuss further cyclic dependency in Section 3.1.1.

The structure of packages and classes of the analysed Java code makes the built architecture. This physical structure makes the initial design information, which I use to realise the architecture of the system, to define the logical architecture.

The **target architecture** is a logical architecture that the corporation or open source community have designed and intended as the project plan. The intention was that the built architecture should be exactly as what has been designed.

The **logical architecture** in the context of my approach is a conceptually defined architecture for a subject system which allows us to define higher levels of abstractions for the built architecture, to impose architectural rules and guidelines of certain architecture paradigm used as a reference, and to conduct the conformance analysis. However, defining logical architecture does not attempt to make any physical change to the recovered built architecture.

The second process is the conceptual re-modularisation by which the hierarchically arranged logical categories are derived. Deriving the categories depends on and respects the built architecture. For example, the original interdependencies among classes and packages should not be changed. More explanation will be given in Section 3.2.

To third process is the imposition of an architecture paradigm through defining logical architecture. This process needs three input data: an architecture paradigm chosen by the architect, the built architecture and the hierarchically arranged logical categories. And the result of this process is naturally a logical architecture of the system implementing the chosen architecture paradigm guidelines as reference.

Finally the fourth process, ascertains the built architecture by differentiating it against the chosen reference architecture paradigm. This process is driven by the logical architecture defined which embeds the reference paradigm guidelines and rules.

The output is the final shape of the logical architecture employing the chosen architecture paradigm, plus two more deliverables: a warning list and reengineering suggestions. The warning list includes finger points to the lines of code, packages and classes that involve in breaching the architectural constraints suggested by the reference architecture paradigm, if any. For packages prevent modularity or contribute in architectural erosion the reengineering suggestions declare issues to be addressed later on according to the situation. This is described further in the last section of this chapter and within the results of the experiment in Chapter 4.
3.1 The Recovery of Built Architecture

The built architecture is what the source code possesses exactly with its physical structuring of classes and packages and their interdependencies. It is fully tool-supported using Sonargraph. The architecture recovery in the context of this thesis takes Java software source code. As motivated in Chapter 1 the architecture modularity may be threaten by some intercommunication patterns happen among classes and packages of the source code. One kind of such patterns is called cyclic dependency which I discuss in the following section.

3.1.1 Architecture Modularity and Cyclic Dependency

Typically, in Java programming language, each source file (.java) has one top-level type. A type in this sense is an object oriented class or any variation of its lexical stereotypes, including interface type, abstract class type, and enum type. Dependencies between source files exist as a consequence of some collaboration among the types in these source files. If two classes in separate source files collaborate with outgoing and incoming dependencies then the cyclic dependency exists as illustrated in Figure 3.1. Moreover, transitive dependencies among several classes, two or more, can yield in cyclic dependencies. For example, class C1 depends on C2 and C2 depends on C3 and C3 depends on C1, and the chain can have arbitrary number of involving classes.

A package in the object oriented paradigm is a set of related classes making a physical unit which is logically cohesive [30]. Cyclic dependencies in object oriented systems exist enormously making large number of classes involve in big cycles [38]. The cyclic dependencies among classes distributed in two packages or more make these packages involve in cycles, too. Nevertheless, two packages can be in a cycle while the distributed classes are not in any cyclic dependency as illustrated in Figure 3.2.
As motivated in the introduction, cyclic dependencies are born by the nature of an object oriented software system. To fulfill a particular feature’s objective of the system a group of related classes may collaborate making these classes involve in different cycles. That is “dependencies between the classes of an object oriented system are a natural consequence of modularisation” [38]. Such kind of object oriented system intercommunication patterns make the modules of the system and at the same time they may be the reason of breaking the modularity in case they are implemented wrongly. We say the architecture is modular if we can see related classes are in distinguishable abstract modules and the packages should be also observed in distinguishable levels as illustrated in Figure 3.3.

Packages are one level higher abstract modules. And this is the highest level of abstraction we can see in source code. Related packages should also make distinguishable higher abstract modules. The overall structure then can be hierarchical, making the built architecture of the system. Usually, the software source code is implemented with the target architecture in mind, by defining classes and packages and their interdependencies. Packages make a physically cohesive unit can be regarded as related packages and therefore they can be collected into modules of a higher abstract level [30]. This permits to perceive the modularity of the system in higher level of abstraction.

But why it is so important to have modular hierarchical structure of the system architecture? The answer is that simply because maintaining modularity and hierarchy is a key to maintain high quality architecture, not erode, not in debt. Software architects strive to save the architecture of their systems from erosion. And as motivated in the introduction, the quality of software system is influenced by the quality of its architecture. More and more Cyclic dependencies may make the modularity of the system dissolves and the architecture faints away. This erosion accumulates a little by little, during the evolution of the system, which drives the built architecture away from the target architecture (or from what it is said to be the correct) and builds its debt over time.

Object oriented intercommunication patterns differ in types and strength, for instance, a class may depend on another class by calling only the name of the latter, or by using its
method, or by inheriting its type, or even by complete aggregation or containment. These mentioned types of dependency have different impacts and they are not the same in value. Also, the kind of the different classes that depend on each other also play a factor in changing the value or cost of the intercommunication pattern. For instance, related classes making a cohesive entity could be allowed to have cyclic intercommunication to make a local module. In addition, the cost of the intercommunication patterns of the classes varies according to where these classes reside in their containing packages and to the relativeness level of the containing packages. In the literature, I found the concept of the intrinsic interdependency which is, as defined by Lakos: “an inherent coupling in the interface of related abstractions” [30]. In previous study conducted by Melton and colleagues, the concept was employed to differentiate between the necessary and unnecessary cyclic dependencies among classes [38]. The researchers also derived some reasoning about cyclic dependencies in the package level. Table 3.1 lists each dependency that can be found among Java classes and whether it is an intrinsic interdependency or not. I found the study is applicable to my research to derive reasoning about the intercommunication patterns in higher abstract levels of system architecture. And this will help me while reasoning about the results after imposing an architecture Paradigm. I leverage this discussion in Chapter 4.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
Dependency & Usage & Intrinsic interdependency \\
\hline
Filed type & Declare as data member & Yes \\
\hline
Return type & Return type of method signature & Yes \\
\hline
Interface call & Invoke interface method & No \\
\hline
Inheritance & Extend super type & No \\
\hline
Special call & Invoke class constructor & No \\
\hline
Implementation & Implement interface & No \\
\hline
Parameter type & Use as method formal argument & Yes \\
\hline
Static call & Invoke static method & No \\
\hline
Uses new & Make instance of type & No \\
\hline
Non-static call & Invoke method of any type but not interface & No \\
\hline
Read & Read static field value & No \\
\hline
Type casting & Cast to type in the hierarchy & Yes \\
\hline
\end{tabular}
\caption{Various dependencies, their usage specifications and characteristics, established by Java classes. I concluded this table from the published work by [38].}
\end{table}

3.2 Conceptual Re-modularisation

As motivated in Chapter 1, the recovery of the built architecture of system source code cannot recover the global architecture of the system. The top abstract level of the built architecture is the package level. It is still not sufficient to be able to differentiate the built
architecture against the target architecture or a reference architecture paradigm. But still one major concern of this study is to guide the architect from the built architecture recovered to the final architecture conformance analysis by maintaining the linkage across the abstract levels from the low level to the top level. Therefore, to take the built architecture to higher levels of abstraction, I introduce the conceptual re-modularisation mechanism.

The conceptual re-modularisation process works in two dimensions to find out modules of higher abstract levels. The first dimension is in the class level to hierarchically categorise the contained classes within each package. The second level of the re-modularisation process works to categorise the packages of a subject system into higher levels to facilitate defining a logical architecture for the subject system. In the class level, the cyclic and non-cyclic dependencies of the classes, of each package, are listed using Dependency Structure Matrix (DSM). Then, the cycles will be used for the algorithm of the re-modularization. Related classes with cyclic dependencies should be in one logical category. The logical categories are hierarchically arranged and related by using the non-cyclic dependencies of classes resulted to belong to different categories.

In the package level, basically, packages involve in cyclic dependencies are assigned to separate higher level logical categories. During this process the original design documents may help partially to understand what kind of role or job each class and package makes. And the re-modularisation activity expose the architect to figure out what was the original intention of the architects of the subject systems to assign each package to a suitable abstract logical category. In the following two sections, I describe each level of the conceptual re-modularisation.

### 3.2.1 Class Level Re-modularisation Algorithm

**The input** of the class level re-modularisation algorithm is a set of source code classes belong to a single physical package, and their interdependencies information. These information can be found by the view of the built architecture explored by Sonargraph. **The output.** The algorithm logically partitions each package into a number of logical categories. Different categories should not include classes involve in a cyclic dependency. Each package will have logical categories with distinguishable levels.

Algorithm Start.

For each package of the subject system.

To make the first logical category:

1. Add all the top-level classes that do not have incoming dependencies.
2. Add all the top-level classes that involve in cyclic dependencies with the classes resulted from step 1. Note: consider only the cyclic dependencies of size 2 and 3 classes.
3. Add to them all their inner classes.
To make a logical category of level \( N \):

- Add all the top-level classes that are directly referenced by the classes of logical category \( N - 1 \). Note: Any class belongs to logical category \( N \) level should not reference back any class of logical category \( N - 1 \) level.
- The same as step 2 above.

Repeat the process for logical category \( N + 1 \) level until all the classes of the subject package are covered.

Algorithm End.

### 3.2.2 Package Level Re-modularisation Rules

**The input.** The package level re-modularisation process takes a set of source code packages and their interdependencies information. These information are visible in the built architecture of the subject system and can be explored by Sonargraph.

**The output.** The algorithm results in logically grouped packages. Figure 3.4 may help to visualise the key idea. The logical categories of the packages are related with non-cyclic dependencies derived from the packages dependencies which are originally made by their classes. The process has two main rules as follows.

![Figure 3.4](image)

**Figure 3.4:** A conceptual re-modularisation process takes as input packages and their classes. By using the algorithm and rules described I realise the logical categories that can show the modularity of built architecture.

1. Related packages. Packages make a cohesive unit are assigned to one logical category, especially when they involve in a cyclic dependency.

2. Unrelated Packages. In this case the cyclic dependency is absolutely unacceptable, therefore such packages should not belong to one logical category.
Generally, all packages involve in cyclic dependencies are related packages. But if the architect discovers after investigating the source code or from the original documents that such packages do not have a related role, then they reside in separate categories. Such packages will be candidate for the list of reengineering suggestions, as partial final output of the process.

### 3.3 Imposing Reference Architecture Paradigm

In order to impose a reference architecture paradigm, a logical architecture for the subject system should be defined, because usually architecture paradigms are defined in higher abstract levels than the recovered built architecture abstract level. The logical architecture acts as a medium to view, manage and analyse the built architecture in higher levels of abstraction. It should bridge the architecture recovery and the architecture conformance analysis. The logical categories for classes and packages are prepared and will be used to define the logical architecture. And then, the guidelines of the chosen architecture paradigm will be defined in the logical architecture. The result is a medium in high abstract level that holds both the built architecture with its logical categories and the reference architecture paradigm at the same time. The architecture paradigm can be chosen by the architect to suit the nature of the subject software system architecture. The process works in two main phases as follows.

#### 3.3.1 Defining a Logical Architecture

The logical categories resulted from the conceptual re-modularisation process can be defined as architectural artefacts, including layers and subsystems. First, each package is assigned to an architectural layer. Then the classes of the package are assigned to the layer. After that, the logical categories of the classes are defined as subsystems in the layer. The architect maps the classes to their subsystems as specified by the logical categories. Defining the dependency among layers and subsystems is also guided by the output of the conceptual re-modularisation process. By now, the initial logical architecture is defined for the subject system. The next phase describes how to impose a reference architecture paradigm.

#### 3.3.2 Implementing Architecture Paradigm Guidelines

Certain architecture paradigm is imposed by implementing its guidelines and rules within the initial logical architecture defined. Usually, an architecture paradigm is described in a set of several layers and their protocol of interdependencies. One or more logical categories of packages make the architecture paradigm layers. Here, the architect investigates where categories of the package level should be assigned. Implicitly and automatically, the categories of the class level should follow their packages.

Any architecture paradigm and can be enforced which may suit the subject system. For example, some architects prefer to have the architecture to be consists of User Interface
(UI) layer, Logic layer and Data Layer. Another example, consider the architecture for a system that has the capability to synchronise a work group among clients. This system has a package or more that deals with the network and server playing the faade role. In this case, one more layer is dedicated to such kind of packages. The defined logical architecture has two main characteristics as follows.

1. Go beyond the built architecture in the abstract level. That is the logical architecture makes the global view of the system architecture. It allows the architects to see the subject system in its higher levels of abstraction. These abstractions make the architecture absolutely manageable and understandable.

2. Embed the guidelines and rules or constraints of the chosen reference architecture paradigm. That is the defined logical architecture draws criteria for evaluating the built architecture.

3. Maintain conceptual modularity. The logical architecture allows only the necessary dependencies that exist in the built architecture, whereas the cyclic dependencies will be visible to the architect in high level of granularity. They are located and linked to their lower level sources, including the packages and classes.

3.4 Ascertainment of the Built Architecture

By now, the architect has the material and guidance to ascertain the built architecture, through differentiation among the three architectural mediums as illustrated in Figure 3.5. By defining the logical architecture through specifying only the dependencies resulted from the conceptual re-modularisation and by implementing the chosen architecture paradigm structure, and its layers protocols, any dependency that exists in the built architecture among the classes and packages that is not specified in the logical architecture will be candidate for concrete conformance analysis.

The architect should find a reference architecture paradigm that suits the nature of the subject system. The sum of all the three architectures make the input of the ascertainment analysis process, namely, the built architecture, the logical architecture defined and the reference architecture paradigm. So far, the architecture paradigm is imposed and implemented in the logical architecture. However, not fully implemented because of the deficiency of integrated methods. Therefore, we keep an a eye on the guidelines and criteria of the referenced architecture paradigm during this process. The built architecture ascertainment process is illustrated in Figure 3.5. In the experiment of this study, I use the MVC architecture paradigm because of the nature of the subject systems of my experiment. However, the process is not specifically tailored to the prescription of MVC paradigm. Hence, I introduce the paradigm in the following section.
3.4.1 MVC Architecture Paradigm

The MVC architecture paradigm has been applied and employed widely [26, 42, 18, 9]. It is worth to use it as sample architectural reference for the experiment. I consulted the literature to understand the original MVC paradigm. I found in average many researchers agree on the following criteria:

**MVC Criterion 1** Model layer must not depend on Controller.

**MVC Criterion 2** Controller layer can depend on Model and View layers.

**MVC Criterion 3** View layer can depend on Controller and Model layers.

**MVC Criterion 4** Model layer notifies View layer through messages.

By looking at Figure 3.6, one may understand that classes belong to Controller and View layers can invoke methods of classes belong to Model layer. However, classes belong to View layer cannot invoke methods of classes belong to Controller layer. Plus, classes belong to Model layer cannot invoke methods of classes belong to View layer. These guidelines draw some criteria to check the conformance of JHotDraw architecture to the MVC pattern. Moreover, Burbeck suggests that Model layer is unaware of the existence of View and Controller layers, and it should respond and change its internal type’s state according to the order of View and Controller layers [9]. In this sense, I do not expect that Model should make method invocations of any type belong to the other two layers. And in the top level the architect is dealing with only three abstract modules, namely the Model, View and...
Controller. In each layer a subset of source packages reside. Now, the architect can check the target architecture whether conforms to such architecture pattern guidelines. Yet, the process in manual but is tolerable.

Figure 3.6: A description of MVC architecture paradigm and its guidelines found in the published work of: [18].

Basically, the conceptual re-modularisation and the enforcement of an architecture pattern, such as the MVC architecture pattern during defining the logical architecture, shows which part of the subject system does not adhere to the modularity concept, for instance, locates violations at their existence where the architect can see why such violation is not acceptable and to what extent.

Moreover, Sonargraph checks accordingly the built architecture whether exhibits any violation to the abstract modules being defined. Sonargraph supports the conformance analysis during defining the logical architecture but not in the final output result of the proposed approach. And the proposed approach does not rely on Sonargraph for the final output results. We will see in the case study how Sonargraph fails to distinguish between necessary and unnecessary dependencies.

### 3.4.2 Warning List and Reengineering Suggestions

I give reasoning about the warnings which result from the last phase of the process and mention about the warnings that Sonargraph highlights when cyclic dependencies occur among source packages.

One purpose of the proposed approach is to guide the architect in conducting architecture conformance analysis through the described processes. The approach does not attempt to modify the built architecture of the system as mentioned above. However, to finalise the output of the approach and link the overall process with the subsequent reengineering activities in a software evolution process, I include reengineering suggestions. These suggestions stemmed from the observation and reasoning about and analysis of a subject system as follows.
I. Package Dimension

For Packages involve in cyclic dependencies.

1. If the packages make a related cohesive unit can remain in the same abstract module but their cyclic dependencies should be revised.

2. If the sum of all types contained by each package do not exceed the maximum number of types that is acceptable to be contained in a single package then the packages should be merged in a single package. Best practice suggests a package may not contain more than 50 top level classes [30, 25]. Otherwise, the packages should be revised.

3. If the packages do not make a cohesive unit then they should not belong to the same abstract module.

4. If a package contains more than 50 classes then the package should be split off.

II. Abstract Module Dimension

According to a certain architecture paradigm referenced, the abstract modules should depend on each other only by the same way the architecture paradigm suggests. For instance, in MVC paradigm: Model layer should not depend on Controller layer. If there is any outgoing dependency from Model layer to the Controller layer in the source code, then the dependency should be cut.

In general, both the warning list and reengineering suggestions result from the architecture conformance analysis process as reasoning about each highlighted architectural breach made by the built architecture, i.e, the source code. The following chapter, gives concrete examples about the deliverables of the process.
Chapter 4

Experiment

The experiment conducted in this study is unfolding. Both an architect and software tool analysis tools collaborate for different activities.

In the next section, I introduce the necessary tools used for the experiments. I describe the situation of the subject systems and the procedure briefly in Section 4.2. Then, I explain the first case study in Section 4.3, and the second case study in Section 4.4. Each case study begins by describing the subject system and ends with the results and discussion. I also describe what factors might threaten the validity of the findings in Section 4.5. And finally in Section 4.6, I validate my findings through a comparison activity with two previous studies.

4.1 Necessary Tools

I mainly used two different tools. The first is Sonargraph [24] which automates the static analysis to recover some information about the built architecture from software source code. And, the second is Microsoft Visio [40] providing an environment to draw freely my own realisations. My supervisor suggested me a Design Pattern Recovery tool [52] to see if such information can aid my research. I applied the tool on JHotDraw Framework and Standard packages which resulted in 56 design pattern instances, while Riehle recovered 20 design pattern instances in the two packages of JHotDraw. Basically, I was able to enumerate 12 of them by investigating the original design documents.

Besides, and for a couple of times, I tried the triangulisation algorithm tool [54] to validate my results of the cyclic dependency detection and classes and packages categorisation. The tool succeeded with small amount of elements, however, it failed when I input the second case study packages structure. It showed that the whole system in only one level. In the following sections I describe the Sonargraph and Visio.
4.1.1 Sonargraph

In this section I introduce Sonargraph-Architect in the context of the thesis. Different features of Sonargraph that are not applicable to my case study are not described. Sonargraph is a software architecture recovery, management and quality measurement tool which exploits static analysis [25]. It visualises the built software architecture and facilitates defining and enforcing logical architecture rules. It also checks the built architecture conformance to the defined logical architecture. Violating code lines will be marked with error markers. The tool detects and visualises cyclic dependencies in the package level and in the class level and computes the ratio of packages involve these dependencies. It displays a summary of some statistics about the analysed system in a view called Dashboard. The tool handles Java source code and uses compiled Java .java with .class or with .jar code files. It comes with three kinds of licenses. Two of them are free and the third is commercial. The free licenses are the evaluation license for 15 days and the free community license which can last for several years but is limited to software projects with less than 50,000 bytecode instructions or approximately 20,000 lines of code. I first used Sonargraph with the evaluation license and then I applied for the community free license and that was fine for my case study with JHotDraw 5.1. However, for JEdit 4.4.2 I needed a license that allows using Sonargraph on such a large system. My supervisor ordered a commercial license for 1,000,000 bytecode instructions and that I was able to conduct an experiment on JEdit 4.4.2. Sonargraph shows the recovered architecture after parsing the source code in the Exploration view in terms of packages and classes with arcs representing the outgoing and incoming dependencies. Sonargraph also enumerates the interdependencies of choosing packages and classes in a separate view called Dependencies.

4.1.2 Microsoft Visio

Visio is a diagramming tool facilitates free drawing by providing necessary illustrating objects for several purposes, especially UML notations. The tool does not have a mechanism to check the correctness of the diagrams and the relationships defined by the user, and does not provide any facility for defining certain software architecture. However, I used it during my experiment to visualise the logical categories and study them by myself to realise and study my findings and then define them in Sonargraph.

4.2 Subject Systems and Procedure

As subject systems, JHotDraw 5.1 and JEdit 4.4.2 are chosen. Table 4.1 summarises some statistics about the source code of the two subject systems.

Before attempting my experiment I have had no knowledge about JHotDraw and JEdit. Both systems are new for me and the architecture recovery reminded me with my former experience in software industry, however, with the guidance of my supervisor this time I did things more professionally in scientific fashion.
<table>
<thead>
<tr>
<th></th>
<th>JHotDraw 5.1</th>
<th>JEdit 4.4.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Bytecode instructions</td>
<td>26,799</td>
<td>290,200</td>
</tr>
<tr>
<td>#Lines of code</td>
<td>8,419</td>
<td>101,490</td>
</tr>
<tr>
<td>#Packages</td>
<td>11</td>
<td>29</td>
</tr>
<tr>
<td>#Classes</td>
<td>173</td>
<td>1,157</td>
</tr>
</tbody>
</table>

Table 4.1: Statistics about the source code of both JHotDraw 5.1 and JEdit 4.4.2 subject systems.

For each subject system, the introduced process is applied, as described in Chapter 3, and the experiment is fivefold:

1. Recover the built architecture of the subject system.
2. Re-modularise the classes and packages conceptually.
3. Define the initial logical architecture according to the logical categories resulted from the conceptual re-modularisation process phase.
4. Impose the chosen architecture paradigm by implementing its guidelines in the defined logical architecture.
5. Collect the intercommunication patterns that are not defined in the logical architecture but the built architecture makes. Concrete analysis is done upon these differences resulting in the final output of each case study.

### 4.3 Case Study on JHotDraw 5.1

JHotDraw is a framework for building graphical drawing editor applications [42]. It is famous among researchers because of its mature architecture. It is rich with design pattern implementations. Its Java source code is publically available with design documents come with the implementation. Many researchers have studied JHotDraw and extracted different views of its architecture, discovered implemented design patterns, and recovered some other design information about it including [1, 47, 16, 34, 53].

The case study starts by recovering the built architecture of JHotDraw 5.1 using Sonargraph. Then I conceptually re-modularise the recovered built architecture of JHotDraw. This makes a partial input for defining a logical architecture for JHotDraw through which the MVC architecture paradigm is enforced. It is known that the built architecture of JHotDraw implements the MVC architecture paradigm. Finally, I ascertain the built architecture of JHotDraw. In the end of the case study, I discuss the JHotDraw system compliance with the MVC paradigm guidelines.
4.3.1 Recovering JHotDraw Built Architecture

Because of the existence of design documents, I studied the architecture of JHotDraw consulting these documents. JHotDraw 5.1 source code consists of 11 packages. The two most important packages are Framework and Standard. These two packages make the architecture of the system. Framework package mainly consists of interfaces as fourteen interfaces and four classes. Understanding the architecture of JHotDraw through the interfaces was straightforward. Standard package is the implementation of Framework package and has the hierarchy of many classes that play the main roles in making the JHotDraw features. After I built my first hypothesis about the architecture of JHotDraw, I used Sonargraph to verify my work.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Effort[Man-hour]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual architecture recovery of JHot-Draw Framework package</td>
<td>41</td>
</tr>
<tr>
<td>Verification of the manual architecture recovery of Framework package using Sonargraph</td>
<td>10</td>
</tr>
<tr>
<td>Tool-supported architecture recovery of JHotDraw Standard package</td>
<td>16</td>
</tr>
<tr>
<td>Manual effort invested along the tool-supported architecture recovery of Standard package</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 4.2: The effort invested to recover the built architecture of JHotDraw 5.1.

In summary, by comparing my work with Sonargraph analysis, I found that four dependencies had been missed and two dependencies were extra. Table 4.2 summarises the effort of the activity. And the final class model of Framework package recovered is shown in Figure 4.1. Since I knew I will continue to find the more abstract relations for the classes I proactively arranged the classes of Standard package in layers as the order of their dependencies flow as shown in Figure 4.1.

4.3.2 The Conceptual Re-modularisation of JHotDraw Built Architecture

Still higher abstract levels are needed to show the architecture of JHotDraw. By studying the main flow of dependency of Framework package one can notice that the dependency flows generally from left to right. The main interfaces are DrawingView, Drawing and Figure. Drawing View depends in Drawing and Drawing depends on Figure, and DrawingView depends also on Figure, but never vice versa. This fact made me see the class model as shown in Figure 4.3, when the three hypothetical logical categories were born, namely Drawing-View, Drawing and Figure categories. The conceptual re-modularisation for Standard package was a little bit harder than it is for Framework package. Standard package has 54 types as one interface and 53 classes. It also contains the classes which implement the interfaces of Framework packages and hence I had to link all the dependencies
Figure 4.1: The manual recovery of JHotDraw 5.1 Framework package class model drawn in Microsoft Visio.

Figure 4.2: The manual recovery of JHotDraw 5.1 Standard package class model drawn in Microsoft Visio.
to Framework package. I used two facts about the classes two figure out the categories. First, the role of the classes whether involve in some hierarchy and doing a related job. For example, in Handle category, by naming convention all the classes do some job related to handles of JHotDraw and they have Handle interface in Framework package as super type. Second, the dependency relation, I summed up all of the dependencies in the class level to unify them in one dependency that forms the relation between each two concluded categories. The re-modularisation activity resulted in 9 hypothetical categories as shown in Figure 4.4. If we remove the classes and keep the categories we will see the higher level of modularity as shown in Figure 4.5. So far, the architecture of JHotDraw 5.1 is going to appear. In the next activity the logical architecture will be defined with the support of Sonargraph.

4.3.3 Defining Logical Architecture for JHotDraw

As mentioned above JHotDraw code implements the MVC architecture paradigm. That is the built architecture is already MVC architecture. Therefore, the logical architecture will be defined and the MVC guidelines will be also enforced in the logical architecture. By using Sonargraph I started to build the logical architecture by using the conceptual categories resulted from the previous activity. Then, I assigned the classes and interfaces to each category they belong to. After that, I defined the relations between the categories as shown in Figure 4.3 and Figure 4.5. Also, the categories of Standard package should
Figure 4.4: The initial logical categories of JHotDraw 5.1 Standard package resulted from the conceptual re-modularisation process phase as drawn in Microsoft Visio.

Figure 4.5: The initial logical categories and their dependency relations of Standard package in JHotDraw 5.1.
depend on the categories of Framework package as absorbed from the built architecture which is the original dependencies implemented in the source code. Therefore, defining the dependencies in the logical architecture does not make any new dependency that does not exist in the built architecture. Instead, it is tracing of the all dependencies that exist between the classes and packages of each category.

This activity made two changes to the initial hypothetical categories resulted from previous activity. Plus, it showed the one architecture that combines the two packages. For example, Oriented Handle category shown in Figure 4.5 is not but just variations of the Handle class implementation. In addition, I found that Tool interface in Framework package and its implementing classes in Standard package and all the hierarchy make a reasonable category; hence I made Tool category.

The final categories yielded for Framework package are: Drawing category, Drawing-View category and Figure category and Tool category. And, the final categories yielded for Standard package are: Box-Handle-Kit category, Command category and Tool category and Drawing-View category and Strategy category and Locator category and Handle category and Figure category and Drawing category. The categories as defined in Sonargraph shown in Figure 4.6.

![Figure 4.6: The final logical categories of JHotDraw 5.1 Framework and Standard packages resulted from the Conceptual Re-modularisation and then defined in Sonargraph.](image)

Then, the classes of each package are assigned to the defined categories as guided by the re-modularisation. Figure 4.7 illustrates some of the assigned types to the defined logical categories. The view goes from class dimension to package dimension linking this level of abstraction to a higher level abstraction given by the defined logical architecture. The logical categories view is one level higher in abstraction than the package dimension. To go to a higher level of abstraction, subsets of categories should form a definite module. I defined the MVC architecture layers as layer groups, as one for Model and the second is for View and the third is for Controller. A layer Group is an architectural artefact that is provided by Sonargraph to define a horizontal logical architecture module. This step resulted in the model shown in Figure 4.8. Since it is known that JHotDraw implements the MVC
Figure 4.7: A partial exploration of the logical categories of JHotDraw after the assignment of classes as shown in Sonargraph.

Figure 4.8: The top abstract level of the defined logical architecture of JHotDraw in Sonargraph.
architecture paradigm, it was easy to know which categories should belong to which MVC layer in the logical architecture. I assigned each category to the appropriate architecture layer according to the nature of its role. In the literature many studies recognised which classes of JHotDraw belong to which layer of the MVC pattern. However, I did not rely on previous work’s results for three reasons: First, the conceptual re-modularisation step draws different path and the categories resulted do not exist in any previous study. Second, the aim of the thesis is to ascertain whether the architecture of JHotDraw, which now it is supposed to be the MVC. And thirdly, previous studies conducted analyses on JHotDraw MVC architecture for different goals, whereas my study is concerned with an alternative variation of objectives. I am curious to see how JHotDraw obeys to the MVC paradigm and where it violates, if any.

The final realisation of the extracted categories and their assignments to each layer is summarised in Table 4.3. The logical architecture defined in Sonargraph showing the categories assigned to each MVC layer with their relations is shown in Figure 4.9. Also, the dependency of each MVC layer on the other two layers of the MVC architecture is defined which is also concluded from the relations among the categories assigned to each layer. After defining the MVC layers logically and assign the conceptual categories to each layer, the dependency among MVC layers and among categories is defined as follows.

1. MVC layer level: Two dependencies from Controller layer to Model and View layers; one dependency from Model layer to View layer; and two dependencies from View layer to Controller and Model layers.

2. Logical category level: Non-cyclic dependencies are defined between every two categories in the same MVC layer if the built architecture has originally such dependency. Note that there will be no cyclic dependencies among logical categories since the conceptual re-modularisation phase made sure to partition a source package in hierarchical levels as described in Chapter 3.

3. Sub layer level: In every MVC layer there are two sub layers, namely standard layer and framework layer. I wanted to explicitly differentiate between classes that belong to the source code packages of JHotDraw: Standard package and Framework package.

<table>
<thead>
<tr>
<th>Logical Category</th>
<th>Package (Built Architecture)</th>
<th>MVC Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drawing</td>
<td>Framework and Standard</td>
<td>Model</td>
</tr>
<tr>
<td>Figure</td>
<td>Framework and Standard</td>
<td>Model</td>
</tr>
<tr>
<td>Handle</td>
<td>Standard</td>
<td>Model</td>
</tr>
<tr>
<td>Locator</td>
<td>Standard</td>
<td>View</td>
</tr>
<tr>
<td>Drawing-View</td>
<td>Framework and Standard</td>
<td>View</td>
</tr>
<tr>
<td>Strategy</td>
<td>Standard</td>
<td>View</td>
</tr>
<tr>
<td>Box-Handle-Kit</td>
<td>Standard</td>
<td>Controller</td>
</tr>
<tr>
<td>Command</td>
<td>Standard</td>
<td>Controller</td>
</tr>
<tr>
<td>Tool</td>
<td>Framework and Standard</td>
<td>Controller</td>
</tr>
</tbody>
</table>

Table 4.3: The logical categories appear as assigned to each logical MVC layer and their source packages.
Figure 4.9: The logical architecture layers defined for JHotDraw with the assigned categories shown in Sonargraph.
This explicit definition shows that Standard package depends on Framework package, but Framework does not depend on Standard.

Figure 4.9 illustrates the dependencies of the both levels, namely, the MVC layer level and the logical category level with the arcs representing the dependencies. It also shows the three non-cyclic dependencies defined between standard and framework sub-layers in each MVC layer. The specification of the dependency is that when a higher level abstract module depends on another module, then all substantial lower level modules of the first are allowed to depend on modules of the latter. The explicit logical definition of sub-layers to resemble the source packages aids in the traceability between the built architecture and logical architecture which supports the architecture conformance analysis. Moreover, to see the logical architecture in a different view, Sonargraph has the exploration view which shows the categories with their containing MVC layers and also with the relations in the category dimension as shown in Figure 4.10. The only thing remained is to reason about this architecture view, which is the job of the next phase.

4.3.4 The Compliance of JHotDraw with the MVC Paradigm

The last phase of the process is applied to conclude the conformance of JHotDraw’s built architecture to the original MVC guidelines. As we saw before, the relations among the logical categories are derived from the existent dependencies among classes in the source code. And only non-cyclic defined is defined and consequently allowed. And as described
in Section 4.1 Sonargraph facilitates the conformance analysis by highlighting in markers all the relations in the built architecture that are not defined in the logical architecture. And then, I evaluate each architecture violation by the guidelines suggested by the imposed architecture paradigm criteria. Sonargraph does not detect any violation. However, there are many sensitive information that should be analysed to ascertain the built architecture of JHotDraw.

4.3.5 Results and Discussion

In the logical category dimension, categories do not make cyclic dependencies as shown in Figure 4.10. This means that there are no cyclic dependencies among packages of the same layer or different layers. As shown in Figure 4.10 Controller layer depends on both Model and View layers. And, View layer depends on Model and Controller layers. Finally, Model layer depends on only View layer as shown in Table 4.4.

<table>
<thead>
<tr>
<th></th>
<th>Model</th>
<th>View</th>
<th>Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>*</td>
<td>170</td>
<td>371</td>
</tr>
<tr>
<td>View</td>
<td>23</td>
<td>*</td>
<td>229</td>
</tr>
<tr>
<td>Controller</td>
<td>0</td>
<td>4</td>
<td>*</td>
</tr>
</tbody>
</table>

Table 4.4: A DSM table shows the number of dependencies of each layer on the other two layers of JHotDraw MVC Architecture

I discuss in details the analysis result of each MVC layer of JHotDraw as follows.

I. JHotDraw Model Layer

If we look again at Figure 4.9 we can see that Model does not depend on Controller which satisfies MVC Criterion 1. MVC criteria are listed in Chapter 3. Then, I wanted to find out how Model depends on View layer. Since Model layer should be loosely connected and can only pass an argument to make change notifications for View layer [9], since passing an argument is an intrinsic interdependency as discussed in Chapter 3.

During the assignment of logical categories to MVC layers, I first assigned Handle category to View layer. The fact that Handle interface is in a cyclic dependency with DrawingView interface, as illustrated in 4.11, made me assign all Handle hierarchy grouped logically by Handle category in View layer.

But then, I found that StandardDrawing class which belongs to Drawing category in Model layer makes an instance of and invokes the construction of NullHandle class which belongs to Handle category in View layer. Therefore, Model goes slightly against MVC Criterion 4. I then moved the Handle category to Model layer but I kept only Handle interface in Drawing-View category in View layer since it involves in a cyclic dependency with DrawingView interface. In the end, Model layer makes 23 type dependencies on View layer with 51 references, as the following. 13 references are static call, 1 implementation and 1 read static
II. JHotDraw View Layer

View layer makes 19 type dependencies on Model layer with 170 references as shown in Figure 4.13. Also, View layer makes 4 type dependencies on Controller as follows. StandardDrawingView class which belongs to Drawing-View category in View layer makes an instance of and calls the constructor of DeleteCommand which belongs to Command category in Controller layer as illustrated in Figure 4.14. The second dependency is more acceptable because it is only a message from View to Controller. That is StandardDrawingView class makes seven interface calls and seven returns of Tool interface which belongs to Tool category in Controller layer. The last two dependencies are return types from DrawingView and DrawingEditor interfaces which belong to Drawing-View category in View layer to Tool
Figure 4.13: All the dependencies outgoing from View Layer to Model Layer in JHotDraw.

Figure 4.14: All the dependencies outgoing from View layer to Controller layer in JHotDraw.
interface which belong to Tool category in Controller. The four dependencies are illustrated in Figure 4.14.

III. JHotDraw Controller Layer

Controller layer depends on View layer with 32 type dependencies in 229 reference as illustrated in Figure 4.15. It also depends on Model layer with 29 type dependencies in 371 references as illustrated in Figure 4.16. Controller is allowed to rely on the two other layers according to MVC Criterion 2. For example, AbstractTool class which belongs to Tool category in Controller layer depends on DrawingView interface which belongs to Drawing-View category in View layer with 7 references as follows. One reference is a filed type, and one reference is a parameter type, 4 references are interface call, and the last reference is a return. On the other hand, DrawingView interface depends on Tool interface with only one reference of Return Type dependency. This dependency is reasonable because View should notify Controller which concrete tool is needed by the user to control a particular drawing object.

I also discuss the situation of the built architecture of JHotDraw in both the class dimension and package dimension as follows.

IV. Architecture Reasoning in Class Dimension

The cyclic dependencies exist among classes do not involve in any architecture violations for two reasons. First, the re-modularisation phase has grouped each subset of related classes in a category, especially the classes that have cyclic dependencies. Second, by

Figure 4.15: All the dependencies outgoing from Controller Layer to View Layer in JHotDraw.
investigating the kind of dependencies I found that out of 14 cyclic dependencies there are only 2 extrinsic interdependencies and the other 12 are intrinsic interdependencies. The nature of the last activity of the case study makes by itself the result of the cases study. That is a great part of the output the proposed process intends to bring about as architecture conformance analysis. Plus, the intermediate deliverables that result from the conceptual re-modularisation and logical architecture defining processes are also parts of the case study results. In addition, the introduced process also provides Warning List and Reengineering Suggestions as described in Chapter 3. This section discusses the two last deliverables in details.

V. Architecture Reasoning in Package Dimension

In the package level, the packages of JHotDraw 5.1 do not have cyclic dependencies among each other. In Figure 4.8 the defined non cyclic dependencies between the logical standard layer, which resembles Standard package in source code, and the logical framework layer, which resembles Framework package in source code, in each MVC layer ascertain that Standard and Framework packages are not involve in a cyclic dependency. If otherwise, Sonargraph should have raised an architecture violation marker.

4.3.6 Warning List and Reengineering Suggestions

The built architecture of JHotDraw does not suffer from violations and the cyclic dependencies in the class level are reasonable. There are no cyclic dependencies in the package
level as described above. The packages and classes are (1) cohesive and (2) have moderate sizes so no package merging, package splitting or class refactoring is required.

As mentioned in Chapter 3, the proposed process does not rely on Sonargraph for the final output. However, it is worth note that Sonargraph still shows an architecture consistency warning because it sees the defined logical architecture experiences an architecture violation because of a logical cycle group as shown in Figure 4.17. According to the original MVC architecture guidelines, there is no architecture violation. This supports the claim of the thesis that automated tools and many methods still cannot address the specifications of certain referenced architecture paradigms. The built architecture of JHotDraw implements

Figure 4.17: Sonargraph suggests that the logical architecture defined for JHotDraw experiences architecture violation even after the imposition of the MVC architecture Paradigm and implementation of its criteria.

the MVC pattern. It obeys to the most rules and guidelines of the original MVC pattern. However, it still violates the MVC pattern with 3 violations should be revised with three issues to be addressed:

1. AbstractFigure and Handle: AbstractFigure abstract class which belongs to Figure category in Model layer reads a static value of Handle interface which belongs to Drawing-View category in View layer. This violates that Model layer should only notify View layer through messages.

2. AbstractFigure and RelativeLocator: AbstractFigure makes a static call to RelativeLocator class in Locator category in View layer. This also violates that Model layer should only notify View layer through messages.

3. StandardDrawingView and DeleteCommand: StandardDrawingView which belongs to Drawing-View category in View layer makes an extrinsic dependency on DeleteCommand which belongs to Command category in Controller layer. This dependency triggers doubts whether View needs to make such a strong aggregation by making an instance of a Controller’s class.

In conclusion, the built architecture of JHotDraw adheres well to the MVC paradigm but not absolutely. It violates the reference architecture paradigm imposed which is at the same time implemented. However, the study can be more accurate with more precise descriptions and concrete criteria for MVC architecture paradigm.
4.4 Case Study on JEdit 4.4.2

JEdit is a text editor for programmers [41]. Its Java source code, which is publicly available, consists of 290,200 lines of code 4.1 which is around 10 times larger than JHotDraw 5.1. It is considered a large application and different versions of it are analysed by many researchers [29, 4, 33, 16]. The introduced process once again is applied to this case study with the four activities as in the previous case study: Architecture recovery, conceptual re-modularisation and logical architecture defining and finally differentiating the built architecture to the chosen reference architecture paradigm.

But opposite to the previous case study, JEdit 4.4.2 does not come with enough design documents. However, the architecture recovery activity relied mainly on studying the source code with the support of Sonargraph and manual investigation. Although there is no explicit information that JEdit 4.4.2 implements the MVC architecture, I imposed the architecture pattern to be able to reach the top level of abstraction for the architecture of JEdit.

Finally, the architecture conformance analysis results in the assessment of JEdit built architecture with reasoning about the violations that occur in the architecture level of JEdit system.

4.4.1 The Conceptual Re-modularisation of JEdit Built Architecture

After having the built architecture using the Sonargraph, I scanned the code and tried to find some information in any design documents that might be available. I wanted find out the original intended architecture of JEdit, i.e., the intended architecture by the developers of JEdit. As mentioned above JEdit does not come with design documents. Only Java documents are available but they do not describe about the architecture of the system.

Two main characteristics about the classes of JEdit are (1) each top level class has many internal classes; and (2) there are classes defined inside interfaces’ definitions. The top level classes are mainly large in term of code lines.

The packages are cohesive that is each package has only related classes doing related job. But 18 packages, in the main package org.gjt.sp.jedit, involve at least one cyclic dependency with some other package. The main package has 21 packages as follows. Print, Proto.jeditresource, Help, Menu, Bufferio, Pluginmgr, Options, Indent, Datatransfer, Visitors, Browser, Input, Search, Bufferset, IO, Msg, Gui, Syntax, Textarea, Buffer and Bsh.

In the beginning, I picked randomly Textarea package which has 39 top level classes, among them 16 classes involve in at least one cyclic dependency of size 2, i.e. the cycle of two classes. Since it is my first experience with JEdit it took me around 12 man-hours to logically categories the package internally. The activity resulted in 3 logical categories.

Then I went to Gui package which took me 9 man-hours, although it is larger than Textarea package. Gui package has 75 top level classes. It has also an internal package called Statusbar which has 16 top level classes. The classes of Gui do not involve in cyclic dependency
however, the classes of Statusbar package play a role in the middle between different classes of the container package Gui. That is they together form a cyclic dependency as shown in Figure 4.18.

![Figure 4.18: Sonargraph shows JEdit Gui package internal structure.](image)

After the conceptual re-modularisation, the modularity of Gui package becomes obvious. That is the main package is logically partitioned into two categories. Also, Statusbar package was partitioned into two categories as shown in Figure 4.19. It is worth to note that the conceptual re-modularisation resolved the cyclic dependency between Statusbar and the classes of Gui exist in the built architecture shown in Figure 4.18. The cycle has disappeared. This will appear in the final output of the process as reengineering suggestions.

![Figure 4.19: The resulted logical categories of JEdit Gui Package from the conceptual re-modularisation process phase.](image)

After I became more familiar with the built architecture of JEdit system and with more 12 man-hours I completed the conceptual re-modularisation of the rest of the packages. In average each package was logically partitioned into two to three categories. Bsh package was exceptional which resulted in 8 logical categories because it has 5 internal small packages. I wanted to logically define each of Bsh internal package to see the interdependency among Bsh’s classes and its packages Figure 4.20.

![Figure 4.20: The logical categories resulted from re-modularising Bsh package.](image)
In total 33 man-hours was the effort spent to conceptually re-modularise the built architecture of JEdit 4.4.2. Here, I noticed that most of JEdit packages found cohesive. That is each package contains classes doing a related job with very few exceptions. And all of them can be partitioned logically in levels.

4.4.2 Imposing Architecture Paradigm on JEdit

For each package I defined a layer architecture artefacts using Sonargraph and assigned each package to its layer. Then the logical categories resulted from the previous activity is assigned to each layer. During the activity or the conceptual re-modularisation I was wondering what the architecture of JEdit could be. As mentioned in the first activity of this case study, there is no explicit description about the architecture of JEdit. In the literature, I found, Brühlmann and colleagues extracted and visualised some knowledge about the architecture of JEdit in 4 layers, namely Ui, Logic, Data and Rest layers [8] and showed some violations among the layers. Patel and colleagues found 14 layers for JEdit and specified which source packages belong to them [43]. However, the two studies do not provide the exact architecture of JEdit.

However, I chose to impose the MVC architecture paradigm through defining a logical architecture for JEdit, since I found it suitable for the nature of the system. I defined three layer groups with Sonargraph as I did in the previous case study, similar to Figure 4.7. To assign the logical categories to each MVC layer, I read in the User Guide and API documentation [41] and the source code of JEdit to get an idea of what is the nature and role of each package and class of the system. I relied on the definition and the roles of the methods of each class to find out their nature. And since the classes are in logical categories I concluded to which MVC layer the containing categories should belong.

Then I went through the classes once again and found that categories of the same package should not be in the same MVC layer. For instance Search package has classes that do a job for View layer and other classes do a job for Model layer. Also, in Indent and Buffer packages, some classes do a job for Controller layer and others do a job for Model layer. I assigned the categories that contain those classes to the appropriate MVC layer. In addition, distinguishing the packages that contribute in View layer was the easiest. That is, it was clear to study the super types of their contained classes whether are of UI nature, for instance, classes that extend, or inherit, directly or indirectly through other JEdit types, GUI types from external packages such as Java Swing. And because the packages of JEdit are cohesive internally this also supported the hypothesis about the UI packages.

Table 4.5 and continued in 4.6 summarises for each MVC layer the assigned logical categories as in their sub layers and the source packages. Finally, the resulted assignment of packages to categories is, Controller layer has logical categories of 7 packages, View layer has categories of 8 packages including some types of the main package org.gjt.sp.jedit, and Model layer has categories of 9 packages as shown in Table 4.7.

As I did with JHotDraw to explicitly link the logical categories to their original package sources I made sub layers with the name of each package. That is every logical category assigned to an MVC layer is also enclosed with a sub layer with a name resembles the source
<table>
<thead>
<tr>
<th>Sub layer :: Logical Category</th>
<th>Package</th>
<th>MVC Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer set :: Ctgr1</td>
<td>org.gjt.sp.jedit.bufferset</td>
<td>Controller</td>
</tr>
<tr>
<td>Buffer set :: Ctgr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visitors :: Ctgr1</td>
<td>org.gjt.sp.jedit.visitors</td>
<td></td>
</tr>
<tr>
<td>Visitors :: Ctgr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visitors :: Ctgr3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer :: Ctgr1</td>
<td>Portion of org.gjt.sp.jedit.buffer</td>
<td></td>
</tr>
<tr>
<td>Indent :: Ctgr1</td>
<td>Portion of org.gjt.sp.jedit.indent</td>
<td></td>
</tr>
<tr>
<td>Input :: Ctgr1</td>
<td>org.gjt.sp.jedit.input</td>
<td></td>
</tr>
<tr>
<td>Input :: Ctgr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Print :: Ctgr1</td>
<td>org.gjt.sp.jedit.print</td>
<td></td>
</tr>
<tr>
<td>Print :: Ctgr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proto.jeditrespirce :: Ctgr1</td>
<td>org.gjt.sp.jedit.proto.jeditresource</td>
<td></td>
</tr>
<tr>
<td>Proto.jeditrespirce :: Ctgr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Help :: Ctgr1</td>
<td>org.gjt.sp.jedit.help</td>
<td>View</td>
</tr>
<tr>
<td>Help :: Ctgr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Help :: Ctgr3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Menu :: Ctgr1</td>
<td>org.gjt.sp.jedit.menu</td>
<td></td>
</tr>
<tr>
<td>Menu :: Ctgr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options :: Ctgr1</td>
<td>org.gjt.sp.jedit.options</td>
<td></td>
</tr>
<tr>
<td>Options :: Ctgr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plug in Manager</td>
<td>org.gjt.sp.jedit.pluginmgr</td>
<td></td>
</tr>
<tr>
<td>Browser :: Ctgr1</td>
<td>org.gjt.sp.jedit.browser</td>
<td></td>
</tr>
<tr>
<td>Search :: Ctgr1</td>
<td>Portion of org.gjt.sp.jedit.search</td>
<td></td>
</tr>
<tr>
<td>Gui :: Ctgr1</td>
<td>org.gjt.sp.jedit.gui</td>
<td></td>
</tr>
<tr>
<td>Gui :: Ctgr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gui :: Statusbar Ctgr1</td>
<td>org.gjt.sp.jedit.gui.statusbar</td>
<td></td>
</tr>
<tr>
<td>Gui :: Statusbar Ctgr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The rest of types</td>
<td>Portion of org.gjt.sp.jedit</td>
<td></td>
</tr>
<tr>
<td>Text area :: Ctgr1</td>
<td>org.gjt.sp.jedit.textarea</td>
<td></td>
</tr>
<tr>
<td>Text area :: Ctgr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Text area :: Ctgr3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: The logical categories resulted and assigned to each MVC layer in the defined logical architecture for JEdit.
<table>
<thead>
<tr>
<th>Sub layer :: Logical Category</th>
<th>Package</th>
<th>MVC Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data transfer :: Ctgr1</td>
<td>org.gjt.sp.jedit.datatransfer</td>
<td>Model</td>
</tr>
<tr>
<td>Data transfer :: Ctgr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data transfer :: Ctgr3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indent :: Ctgr2</td>
<td>Portion of org.gjt.sp.jedit.indent</td>
<td></td>
</tr>
<tr>
<td>Buffer :: Ctgr2</td>
<td>Portion of org.gjt.sp.jedit.buffer</td>
<td></td>
</tr>
<tr>
<td>Syntax :: Ctgr1</td>
<td>org.gjt.sp.jedit.syntax</td>
<td></td>
</tr>
<tr>
<td>Syntax :: Ctgr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Syntax :: Ctgr3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search :: Ctgr2</td>
<td>Portion of org.gjt.sp.jedit.search</td>
<td></td>
</tr>
<tr>
<td>Buffer IO :: Ctgr1</td>
<td>org.gjt.sp.jedit.bufferio</td>
<td></td>
</tr>
<tr>
<td>IO :: Ctgr1</td>
<td>org.gjt.sp.jedit.io</td>
<td></td>
</tr>
<tr>
<td>IO :: Ctgr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Msg :: Ctgr1</td>
<td>org.gjt.sp.jedit.msg</td>
<td></td>
</tr>
<tr>
<td>Msg :: Ctgr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bsh :: Classpath</td>
<td>org.gjt.sp.jedit.bsh</td>
<td></td>
</tr>
<tr>
<td>Bsh :: Collection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bsh :: Commands</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bsh :: Reflect</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bsh :: Parser &amp; Interpreter</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bsh :: Asm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bsh :: Ctgr1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bsh :: Ctgr2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.6: (Continued) The logical categories concluded and assigned to each MVC layer in the defined logical architecture for JEdit.
package's name. Moreover, the dependencies among the MVC layers and the categories are defined as follows.

1. MVC layer level: Two dependencies from Controller layer to Model and View layers; one dependency from Model layer to View layer; and two dependencies from View layer to Controller and Model layers.

2. Logical category level: Non-cyclic dependencies are defined between every two categories in the same MVC layer.

3. Sub layer level: In each MVC layer as described above sub layers are defined which resemble the source packages and link explicitly the logical categories to their source packages through the logical architecture artefacts. Non-cyclic dependencies are defined between every two sub layers.

The final logical architecture defined for JEdit 4.4.2 is illustrated in 4.21. With 4 man-hours the logical architecture was almost completely defined including imposing the MVC architecture paradigm. MVC Model layer defined for JEdit with its sub layers and assigned categories is illustrated in Figure 4.22.

4.4.3 The Ascertainment of JEdit Built Architecture

As mentioned in the previous activity of this case study there is no evidence that JEdit implements the MVC architecture pattern. Assuming that the architecture of JEdit is the MVC pattern the conformance analysis should check how the built architecture maintains a mature architecture definition. Unlike the previous case study, JEdit has many cyclic dependencies among its packages. As mentioned in the first activity of the case study 18 packages involve in at least one of the 47 cyclic dependencies. These cycles resulted because 88 classes distributed in these packages make originated the cyclic collaborations. By looking at 4.23 one can see the red arcs among the sub layers in MVC View layer and Model layer. These red arcs tell that the defined logical architecture does not allow such dependencies and therefore they breach the modularity of the architecture of JEdit. Not all 47 cyclic dependencies appear in red because I defined round trip collaboration between View and Controller and between View and Model layers, so Sonargraph coloured the cyclic dependencies among these layers in only green. Since I allowed two round trip dependencies between View and Controller layers and also between View and Model layers, the cyclic dependencies among packages assigned to different layers are not shown as violations. This is because Sonargraph does not provide concrete specifications of the type of dependency that can be allowed between two architectural artefacts.

4.4.4 Results and Discussion

I studied in more depth the size and heaviness of the dependencies that go from each layer to the others. DSM Table 4.7 summarises the number of dependencies that each MVC layer makes on the other two layers.
Figure 4.21: The logical architecture defined for JEdit showing each MVC layer with the assigned logical Categories.

<table>
<thead>
<tr>
<th></th>
<th>Model</th>
<th>View</th>
<th>Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>*</td>
<td>3,530</td>
<td>224</td>
</tr>
<tr>
<td>View</td>
<td>713</td>
<td>*</td>
<td>419</td>
</tr>
<tr>
<td>Controller</td>
<td>0</td>
<td>318</td>
<td>*</td>
</tr>
</tbody>
</table>

Table 4.7: A DSM table shows the number of dependencies of each layer on the other two layers of JEdit MVC Architecture.
Figure 4.22: Model layer of JEdit with the logical categories assigned.

Figure 4.23: The final logical architecture defined for JEdit 4.4.2.
From the table we can conclude the ratio of dependencies among the layers as 1:0.75 between Controller and View layers, 1:0.00 between Controller and Model layers and 1:0.20 between View and Model layers.

A red arc between two sub layers means that the dependency exists in the built architecture but not defined in the logical architecture as only non cyclic dependencies are defined among sub layers. Note: the categories of each sub layer are dimmed for clarity. Moreover, and as described in Chapter 3, a dependency on type through an invocation of one of its methods should not be treated as just using its name in the body implementation of in a method’s signature. Therefore, I studied the nature of each dependency happens between each two layers. The study resulted in the points as follows.

1. All of the dependencies outgoing from View layer to Controller layer are not intrinsic. Among the 8 packages assigned to View layer 4 of them depend on 4 packages of Controller layer. Also, View layer contains types in the main package org.gjt.sp.jedit which depend on 5 packages in Controller layer.

2. 92.5 percent of the outgoing dependencies from View layer to Model layer are extrinsic. All the packages of View layer depend on all the packages of Model layer, as summarised in Table 4.9.

3. 88.2 percent of the outgoing dependencies from Model layer to View layer are extrinsic. All of the 9 packages of Model layer depend on all of the packages of View layer, as summarised in Table 4.10.

4. All dependencies outgoing from Controller layer to Model layer are extrinsic. Only 5 packages of Controller depend on 4 packages of Model, as summarised in Table 4.11.

5. 99.18 percent of the outgoing dependencies from Controller layer to View layer are extrinsic dependencies. All of the packages of Controller layers depend on only two packages of View layer and many of the types in the main package org.gjt.sp.jedit, as summarised in Table 4.12.

<table>
<thead>
<tr>
<th>Controller Layer</th>
<th>Options</th>
<th>Search</th>
<th>Gui</th>
<th>Text area</th>
<th>The rest of types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer set</td>
<td>extrinsic</td>
<td>extrinsic</td>
<td>extrinsic</td>
<td>extrinsic</td>
<td>extrinsic</td>
</tr>
<tr>
<td>Visitors</td>
<td>extrinsic</td>
<td>extrinsic</td>
<td>extrinsic</td>
<td>extrinsic</td>
<td></td>
</tr>
<tr>
<td>Input</td>
<td>extrinsic</td>
<td>extrinsic</td>
<td>extrinsic</td>
<td>extrinsic</td>
<td></td>
</tr>
<tr>
<td>Buffer</td>
<td>extrinsic</td>
<td>extrinsic</td>
<td>extrinsic</td>
<td>extrinsic</td>
<td></td>
</tr>
<tr>
<td>Indent</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>extrinsic</td>
</tr>
</tbody>
</table>

Table 4.8: The dependencies characteristics, in terms of intrinsic or extrinsic, outgoing from View layer to Controller layer in the defined MVC architecture for JEdit.

In Table 4.8 the summary of the kind of each dependency outgoing from View to Controller is given. Extrinsic dependency can be one of the following: Interface call, Inheritance, Special call, Implementation, Static call, Uses new, Non-static call, and Read, as described in Chapter 3.

2. 92.5 percent of the outgoing dependencies from View layer to Model layer are extrinsic. All the packages of View layer depend on all the packages of Model layer, as summarised in Table 4.9.

3. 88.2 percent of the outgoing dependencies from Model layer to View layer are extrinsic. All of the 9 packages of Model layer depend on all of the packages of View layer, as summarised in Table 4.10.

4. All dependencies outgoing from Controller layer to Model layer are extrinsic. Only 5 packages of Controller depend on 4 packages of Model, as summarised in Table 4.11.

5. 99.18 percent of the outgoing dependencies from Controller layer to View layer are extrinsic dependencies. All of the packages of Controller layers depend on only two packages of View layer and many of the types in the main package org.gjt.sp.jedit, as summarised in Table 4.12.
Table 4.9: The dependencies characteristics, in terms of intrinsic or extrinsic, outgoing from View layer to Model layer in the defined MVC architecture for JEdit.

<table>
<thead>
<tr>
<th>View Layer</th>
<th>Model Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>intrinsic</td>
<td>extrinsic</td>
</tr>
<tr>
<td>extrinsic</td>
<td>extrinsic</td>
</tr>
<tr>
<td>extrinsic</td>
<td>extrinsic</td>
</tr>
<tr>
<td>extrinsic</td>
<td>extrinsic</td>
</tr>
<tr>
<td>extrinsic</td>
<td>extrinsic</td>
</tr>
<tr>
<td>extrinsic</td>
<td>extrinsic</td>
</tr>
<tr>
<td>extrinsic</td>
<td>extrinsic</td>
</tr>
<tr>
<td>extrinsic</td>
<td>extrinsic</td>
</tr>
<tr>
<td>extrinsic</td>
<td>extrinsic</td>
</tr>
<tr>
<td>extrinsic</td>
<td>extrinsic</td>
</tr>
<tr>
<td>extrinsic</td>
<td>extrinsic</td>
</tr>
<tr>
<td>extrinsic</td>
<td>extrinsic</td>
</tr>
<tr>
<td>extrinsic</td>
<td>extrinsic</td>
</tr>
<tr>
<td>extrinsic</td>
<td>extrinsic</td>
</tr>
<tr>
<td>extrinsic</td>
<td>extrinsic</td>
</tr>
<tr>
<td>extrinsic</td>
<td>extrinsic</td>
</tr>
<tr>
<td>extrinsic</td>
<td>extrinsic</td>
</tr>
<tr>
<td>extrinsic</td>
<td>extrinsic</td>
</tr>
</tbody>
</table>

Bash

Buffer IO

Search

Data transfer

Syntax

Indent

Buffer

IO

Msg

Options

Menu

Plugins in Manager

Browser

Text area

Gui

The rest of types
<table>
<thead>
<tr>
<th>View Layer</th>
<th>Model Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data transfer</td>
</tr>
<tr>
<td>Text area</td>
<td>intrinsic</td>
</tr>
<tr>
<td>Gui</td>
<td>intrinsic</td>
</tr>
<tr>
<td>The rest of types</td>
<td>extrinsic</td>
</tr>
<tr>
<td>Browser</td>
<td>extrinsic</td>
</tr>
</tbody>
</table>

Table 4.10: The dependencies characteristics, in terms of intrinsic or extrinsic, outgoing from Model layer to View layer in the defined MVC architecture for JEdit.

<table>
<thead>
<tr>
<th>Model Layer</th>
<th>Controller Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Buffer set</td>
</tr>
<tr>
<td>Msg</td>
<td>extrinsic</td>
</tr>
<tr>
<td>IO</td>
<td>extrinsic</td>
</tr>
<tr>
<td>Buffer</td>
<td>extrinsic</td>
</tr>
<tr>
<td>Indent</td>
<td>extrinsic</td>
</tr>
<tr>
<td>Syntax</td>
<td>extrinsic</td>
</tr>
</tbody>
</table>

Table 4.11: The dependencies characteristics, in terms of intrinsic or extrinsic, outgoing from Controller layer to Model layer in the defined MVC architecture for JEdit.
Table 4.12: The dependencies characteristics, in terms of intrinsic or extrinsic, outgoing from Controller layer to View layer in the defined MVC architecture for JEdit.

I discuss in details the analysis result of each MVC layer of JEdit as follows.

I. JEdit Model Layer

By looking at Figure 4.22 we can see that Model layer does not depend on Controller which satisfies MVC Criterion 1. However, Model layer makes heavy dependencies on View layer is described in point 3 above. Therefore, Model does not conform to MVC Criterion 4. It has two packages involve in cyclic dependencies locally. All of its packages involve in cyclic dependencies with packages belong to View layer.

II. JEdit View Layer

It is acceptable for View layer to make extrinsic dependency (e.g., method invocations) of classes in Model layer to some extent, as concluded from MVC Criterion 3. However, View layer makes 318 extrinsic dependencies on Controller. All of its packages involve in cyclic dependencies locally and globally with packages belong to both Controller and Model layers.
III. JEdit Controller Layer

Controller depends on Model layer with 224 references and on View layer with 419 references, as it is permitted by MVC Criterion 2. But View layer still makes 3,530 dependency references on Model layer. This may spark doubts about the who is the controlling actor among the three layers.

On the other hand, Controller layer is the only layer that does not have packages involved in cyclic dependencies locally as shown in Figure 4.24. But still 5 of its packages involve in cyclic dependencies with packages belong to View layer.

I also discuss the situation of the built architecture of JEdit in both class dimension and package dimension as follows.

IV. Architecture Reasoning in Class Dimension

In average the classes of JEdit exceeds 500 lines of code and many hit 2000 lins of code. The large classes of JEdit makes it hard to comprehend well about its architecture. As mentioned in the beginning of the case study, top level classes of JEdit have in average many inner classes which make them more complex and contribute in dimming the global architecture of JEdit.

V. Architecture Reasoning in Package Dimension

Most of the packages of JEdit have less than 30 top level classes. Exceptionally, Textarea, Gui and Bsh packages which include 40, 90 and 165 top level classes respectively. Since each package is said to be a cohesive collection of classes, the large collections of classes may make the package incoherent. Therefore, the large packages in JEdit contribute in breaching the modularity of its architecture. This is evident by looking at Figure 4.24. Even after defining the logical architecture and imposing the MVC pattern, after the re-modularisation and enforcing non cyclic dependencies, the architecture of JEdit still suffers from many cyclic dependencies as shown in red arcs in Figure 4.24.

Activity 4 implements the architecture conformance analysis of the proposed process and brings about the reasoning and descriptions of the status of the built architecture of JEdit 4.4.2. After the built architecture of JEdit is recovered, the re-modularisation and logical architecture defining intermediate deliverables make parts of the results. To conclude the results and to link the deliverables to a broader context I describe Warning List and Reengineering Suggestions for JEdit 4.4.2 as follows.
### 4.4.5 Warning List and Reengineering Suggestions

The built architecture of JEdit 4.4.2 suffers from many cyclic dependencies. Some of these cyclic dependencies are necessary as described in Activity 4. For source code written in Java it is not so much of a problem to have packages involve in cyclic dependencies since Java bytecode files depend on the names appear in the constant pool of other bytecode files as the mechanism of Java Virtual Machine [32]. This is from Development aspect, especially, if such packages belong to a single abstract module, e.g. Model layer, then such cycles would be considered locally and do not breach the global modularity of the system.

However, from system comprehension perspective these packages make it hard to understand the system by software architects. And this reflects my experience with JEdit though this case study. Only lexically it was easy to distinguish the packages that I hypothesised would go under View layer, although the packages involve in cyclic dependencies.

As mentioned in Activity 3, since the contained classes more likely inherit classes of visual graphical nature. And even after enforcing the MVC pattern and assigning these packages two one module, namely MVC View layer, the cyclic dependency pervades the modularity of View layer internally and Sonargraph raised the warning with red arcs as shown in Figure 4.26. In class level, understanding top level classes was hard because of the myriad cyclic dependencies they make. Table 4.13 summarises the number of classes and packages involve in cyclic dependencies in JEdit 4.4.2. The provided warning list wants to stress that: “Saying that there is such number of cyclic dependencies in a subject system is not enough”. Sonargraph shows 2,320 warnings and estimated the ratio of packages involve in cyclic dependencies as 72.41 percent as illustrated in Figure 4.25. As mentioned before, this estimation should not be taken in full trust and the reasoning explained in the last activity.

#### Table 4.13: Total number of cyclic dependencies among classes and packages of JEdit 4.4.2.

<table>
<thead>
<tr>
<th></th>
<th>#Classes</th>
<th>#Packages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclic Dependency</td>
<td>194</td>
<td>18</td>
</tr>
<tr>
<td>Intrinsic interdependency</td>
<td>2</td>
<td>N/A</td>
</tr>
<tr>
<td>Extrinsic interdependency</td>
<td>192</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Figure 4.24: Cyclic dependency visualisation by Sonargraph in the Package category Level.
of this case study says that these warnings are not necessarily accurate. At the same time, I reasoned about the status of the intercommunication among packages and abstract modules in details in Activity 4.

In conclusion, the built architecture of JEdit 4.4.2 suffers from many architecture violation instances. There are many packages breach the modularity of the system. But the built architecture is not that erode as estimated by Sonargraph Dashboard in Figure 4.25. And, I still can conclude the following possible enhancement attempts.

1. Revise Gui package to resolve the cycle with the internal package Statubar as shown in Figure 4.19.
2. Revise large classes and split them off.
3. Revise packages making cyclic dependencies when they belong to different layers.

4.5 Threats to Validity

There are numerous factors affect the generalizability of the study findings. Given the described experimental settings and procedure, the following factors should be considered since they might affect the results.

4.5.1 Vagueness of MVC Paradigm Description

I could not find a definite and precise description of each dependency that can be allowed from one layer to the other two layers of MVC architecture paradigm. Therefore, reasoning about the built architecture of both subject systems JHotDraw 5.1 and JEdit 4.4.2 may be affected. Moreover, it is unknown whether JEdit 4.4.2 system implements the MVC architecture paradigm. However, to continue my case study I had to think about a solution to bring about my findings. I attempted the imposition of MVC paradigm during the
activity of defining a logical architecture for JEdit. The enforcement of the MVC layers and dependency rules succeeded in the context of the thesis but cannot be necessarily support a claim that this is the architecture of JEdit 4.4.2.

Finally, I did not comprehend well MVC Paradigm and I did not bring formal study about it. That is, I did not address how reliable it is to reference to MVC in order to both define a logical architecture for a subject built architecture and to ascertain the built architecture.

4.5.2 Impreciseness of Intercommunication Patterns Classification

I exploited the study of intrinsic interdependency to bring about careful reasoning and ascertaining of some intercommunication patterns among system classes, packages and abstract modules, which breach the modularity of the system, such as cyclic dependency. However, there must be deeper and alternative classifications and impact analysis of various intercommunication patterns that may exist in complex large systems.

4.5.3 Subject Systems Representativeness

It is uncertain whether the subject systems chosen for this experiment are representative for real world setting of architecture analysis. JHotDraw 5.1 is very moderate in size but still has a reputation of a mature architecture that is worth to be subjected in research experiments as mentioned in the beginning of its case study. However, the second subject system JEdit 4.4.2 should be large enough to be realistic and make some equilibrium for the study.

4.5.4 Familiarity with the Subject Systems

As mentioned in Section 4.2, I had no former knowledge about the two systems. Being confronted with completely unknown systems is the common situation in software industry when a new comer is assigned the early development tasks with the software project of the organisation. Hence, the findings can still be generalised to real practice situation.

On the other hand, there are still the team members who have built some experience with the organisation’s software. Hence, results cannot be generalised to such a situation.

4.5.5 Architect’s Expertise and Competency Level

I included the cost in terms of man-hour I took to manually recover the architecture of JHotDraw. This is absolutely personal experience and cannot be generalized at all. The time is spent to understand the design pattern instances and cyclic dependencies and transitive dependencies contributed to raise the cost. Although I have applied some design patterns during my career time, I still found it hard to distinguish many instances of the design
patterns implemented in JHotDraw. Hence, results may be different for architect who is specialised in Design Patterns. It is worth also to mention that Riehle’s Role Models documentation helped me to understand the patterns.

4.5.6 Original Design Documents

Once again, the two case studies have extremely opposite circumstances regarding two the original design documents. In the first case study the original design documents of JHotDraw 5.1 were available. However, I could not find the original design documents of JEdit 4.4.2. Perhaps this can give more credentials to my study that I experimented the proposed process in two different situations.

4.6 Comparison with Two Previous Studies

During the former several months, my supervisor handed me two published studies to investigate and comprehend about them. Later on, my supervisor suggested me to compare my work with those two studies. During the time, I built my own understanding about such studies. And when I started to get some obvious results, my supervisor advised me to validate my findings against the findings of the two studies techniques. In this section I introduce the two published studies and provide the results of a comparison activity with them.

4.6.1 Introduction


Riehle’s objective is to extract the relative information about the various “roles” that each class in object oriented software source code plays as a part of the whole roles network that the classes exhibit in variety of intercommunication patterns. The contribution of Riehle’s work is mainly, and according to my personal understanding, twofold. The first one is to provide information about the classes of system source code that cannot be captured by UML Class Diagram, which hinder the architect from observing various, important information exist in the built architecture. The second contribution of Riehle’s work is to capture, project and represent the design patterns instances implemented by the source code which are also hard to be observed or understood by only investigating the plain built architecture, source code or original class diagrams.

Abi Antoun’s objective is to evaluate the conformance of the built software architecture of a subject system to the target architecture. The target architecture, is what the original
architects had intended and designed to be the implementation of the source code, of the software project being developed or evolved. The built architecture is the structure of the source code implemented for the system. The contribution of Abi Antoun’s work is mainly, and according to my personal understanding, threefold. The first is an automated mechanism with a software tool, named SCHOLIA, which recovers the built architecture of object oriented software systems through static analysis which results in visualisations represent potential runtime objects of the source code classes. The visualisations are graphs that represent the objects in hierarchical structure through ownership object abstractions. The second contribution of Abi Antoun’s work is to synchronise the built software architecture with the target architecture whenever the system evolves over time. The third contribution is an automated mechanism to evaluate the conformance of the built software architecture to the target architecture.

4.6.2 Overall Comparison Activity Results

I evaluated my findings with the works of Abi Antoun [1] and Riehle [47] and I conclude the results in the following points.

1. Riehle’s Role-Based Modelling recovers design information that is not explicit and are absent in UML Class Diagrams. However, it is clear that Riehle’s objectives were different from mines. Riehle wants to marry the class design with the role models, as he stated in his PhD thesis. Figure 4.11 illustrates a sample of the Role Models published with his thesis. Therefore, the design documentation provided lies in the horizontal dimension of abstraction. And, it turns out that my technique and Riehle’s technique are not similar in nature, motivations and goals. Each addresses alternative issues about the built architecture.

![Figure 4.26: A portion of the Role-Based model of JHotDraw 5.1 recovered by Riehle in his PhD Thesis.](image)

2. Riehle’s work helps to understand the design patterns implemented by the source code of the subject system. When I was studying the original design documents of JHotDraw, I could not understand well how the implemented design patterns reside across the classes of JHotDraw’s source code. However, after studying Riehle’s thesis, I became very familiar with the instances of the design patterns in JHotDraw.
3. Abi Antoun’s OOGs capture statically the potential intercommunication network of objects that may exist at runtime. An object is an instance of a class at the system’s runtime. The OOGs also recover information about the subject system that UML Class Diagram lacks to capture. Abi Antoun contributes to the level of architecture synchronisation and conformance analysis. Abi Antoun’s tool extracts the OOGs from subject system source code but only after annotating the source code of the subject system.

4. In the PhD thesis of Abi Antoun, I studied the OOGs extracted from a case study on JHotDraw 5.1. In the published JHotDraw OOGs, I found 10 dependencies among the objects found which I was not able to relate them to the built architecture of JHotDraw I recovered. I took the initiative to make more investigation to answer why these objects dependencies exist in the OOGs but I still cannot find them in my class diagrams.

One instance is the cyclic dependency between Tool and Handle as shown by a portion of the published OOGs, as shown in Figure 4.27. There is a box titled: “Tool (+): Tool” which means that the box has collapsed types belong to Tool hierarchy through the ownership containment. From OOG point of view the graph shown in Figure 4.27 says that (1) at least one of Tool hierarchy types depends on Handle interface and (2) at the same time, Handle interface depends on one of the types of Tool hierarchy. The same figure also shows that Handle belongs to Model layer of the MVC architecture of JHotDraw. The original MVC pattern suggests that Model layer must not depend on Controller. Handle interface does not depend on any type of Tool hierarchy as shown in 4.28.
Figure 4.28: All dependencies of Handle Interface in JHotDraw 5.1.
Chapter 5

Discussion and Conclusion

This chapter summarises the contributions and limitations of this thesis and proposes further directions for future research.

5.1 Contributions

This thesis addresses architecture conformance analysis requirements with re-modularisation and differentiation techniques for careful reasoning about software architecture. It introduces a process transforms plain recovered architecture into gradual higher abstract levels maintaining traceability across the abstracts in vertical dimension. And then it differentiates the built architecture against certain architecture paradigm resulting in an evaluation of the architecture. Further, I summarise the contribution in points as follows.

1. Showing how the high abstract level of large system’s architecture can be captured by conceptual re-modularisation

2. Bridging the gap between plain architecture recovery and architecture conformance analysis.

3. Careful reasoning about the communication of architectural artefacts that reflect the internal communication of source code.

4. Vertical abstraction mechanism maintains traceability across different levels of abstraction with gradual pace starting from a recovered built architecture and ending with the top level view of the architecture.

5. Overall guidance for architects to reason carefully about the architecture of a subject system by differentiation against certain well defined architecture paradigm guidelines and rules.
5.2 Limitations

The thesis has shown how conceptual re-modularisation and imposing a software architecture paradigm supports the architecture conformance analysis. The approach was validated with two case studies, however the study has some limitations because of four aspects as follows.

Imposed Architecture Paradigm

In both case studies I employed the MVC architecture paradigm as a reference architecture to conduct the architecture conformance analysis. As described in the experiment MVC paradigm was the most suitable for the nature of the subject systems. Although I did not tailor the proposed technique to suit only MVC paradigm, the overall study still needs to consider alternative architecture paradigms. In Section 5.3 I mention this aspect as a future work.

Programming Language Paradigm

The introduced technique is verified by experiment on two subject systems and both are written in Java language. I discussed in the results of the second case study that the existence of cyclic dependencies among packages does not affect the development activity, such as task allocation among team members, compilation, or unit testing, because the source code of the subject system is written in Java. However, in the context of C++ language, it is still so much of a problem. Compiling the classes of a package that involves in a cycle means recompiling all the classes in the involved packages in the cycle. Therefore, if my experiment had a subject system written in C++ then the analysis and discussion would have differed with a great deal.

Analysis of Object Oriented Intercommunication Patterns

I used the term “intrinsic interdependency” to analyse the cyclic dependencies among classes. I also tried to justify cyclic dependencies in the package level by differentiating the type and size of the underpinning interdependencies made by the classes which reside distributively in different packages. However, more classification can be concluded to reason more carefully about why two packages are in a cycle and whether such cycle is necessary. I can still see that object oriented intercommunication patterns can be studied more deeply.

Integrated Tools Weaknesses

Main weaknesses of Sonargraph does not provide various types of the architectural dependency that can be defined among categories or layers. It also does not have special settings
for any well known architecture paradigm, including MVC and SOA, and etcetera. As a static tool, it does not make considerable false positives. I only saw redundancy in the number of dependencies in the class level sometimes when the dependency is calling an interface method. As I mentioned in Chapter 3, I only focus on the features of Sonargraph that are of the interest to this study.

Microsoft Visio is a helpful UML drawing tool. I drew the figures provided in this thesis, beside the Sonargraph views.

5.3 Opportunity for Future Research

During the research on this topic, a number of ideas and research directions could motivate a future research in the area.

Alternative Architecture Paradigms

There are several software architecture paradigms, including Service Oriented Architecture (SOA).

Alternative Factors Dissolving Architecture Modularity

In the context of this thesis, only cyclic dependencies are investigated among many possible alternative factors that could contribute in dissolving the modularity of system architecture. Hence, among the various aspects that need to be addressed by research in architecture conformance analysis, I list five aspects as follows.

1. What factors, beside cyclic dependency, do contribute in dissolving system modularity?

2. In which level of abstraction, in vertical dimension, the relative information about these factors can be captured?

3. How harmful are they if they play a role in breaching the architecture modularity?

4. What kind of symptoms software architecture may have because of such factors?

5. How can such symptoms aid in identifications to feed an architecture conformance analysis process?

Combination with Dynamic Analysis

It is known that static analysis and dynamic analysis complement each other and raise the accuracy of the final results of an analysis process. Naturally, the static structure of
the system defines the boarders for its behaviour at run time. However, it is also known that a software system may behave and have an internal intercommunication network that cannot be captured by static observations. Hence, I can mention two aspects are worth to be addressed.

1. Does Software System at runtime make an internal intercommunication network that violates its architecture modularity?

2. If so, to what extent the runtime network could violate the architecture modularity, why and how?

5.4 Closing Words

This thesis has contributed to the architecture conformance analysis by providing a reverse software engineering process that combines the abstraction, modularisation and ascertain- ment techniques and integrate these techniques in a process results in fine grained analysis of and careful reasoning about software architecture. The information gathered by the experiments showed where recovered built architectures conform to referenced architecture paradigm guidelines and where they violate. The process does not attempt to modify the subject system but it provides reengineering suggestions for subsequent activities that the architect may commit later on. This thesis is a stepping stone toward support for the early phase of the software evolution process when all subsequent activities of the process depend solely on the accuracy of this phase. I did my utmost effort to contribute in the area of research because I believe it is vital for building high quality software systems. That is, reasoning about the architecture of software system is absolutely reasoning about its quality.
Bibliography


Appendices
A Appendix A

This appendix describes the graphical notation used in the diagrams of this thesis.

A.1 Notation Guide for Sonargraph

Logical category is the architecture abstraction that results from the Conceptual Re-modularisation step of the introduced process. It contains related classes, of a package, that may contribute in some top level architecture layer. It is in a higher level of abstraction than class/package view level.

![Logical category](image)

Figure A.1: Logical category.

Sonargraph facilitates the environment for the architecture to define the logical architecture using the element illustrated in A.1, and then classes of source code be assigned to it as appropriate.

![Sub layer](image)

Figure A.2: Sub layer.

The logical categories of each package are linked to their packages through the sub layer architecture artefact, as illustrated in A.2, provided by Sonaragph.

The architect can assign each package, entirely or partially, to a sub layer.

![Layer](image)

Figure A.3: Layer.

The layers of an imposed architecture paradigm can be defined in Sonargraph with the provided architecture artefact, Layer, as illustrated in A.3. Then sub layers and logical categories can be assigned to each layer as appropriate.

Sonargraph parses Java source code and visualises the recovered packages with the symbol illustrated in A.4.

Sonargraph visualises the recovered classes with the symbol illustrated in A.5.

Sonargraph parses visualises the recovered interfaces with the symbol illustrated in A.6.

Sonargraph visualises the recovered outgoing dependencies from classes and interfaces with the symbol illustrated in A.7, as a green arc meaning the dependency does not violate the
Sonargraph visualises the recovered ingoing dependencies to classes and interfaces with the symbol illustrated in A.8, as a green arc meaning the dependency does not violate the architecture.

Sonargraph visualises the recovered outgoing dependencies from classes and interfaces with the symbol illustrated in A.8, as a red arc meaning the dependency violates the architecture.

Sonargraph visualises the recovered ingoing dependencies to classes and interfaces with the symbol illustrated in A.10, as a red arc meaning the dependency violates the architecture.
Figure A.6: Source code interface.

Figure A.7: An outgoing dependency exists in the source code which does not violate the architecture.

Figure A.8: An ingoing dependency exists in the source code which does not violate the architecture.

Figure A.9: An outgoing dependency exists in the source code but violates the architecture.

Figure A.10: An ingoing dependency exists in the source code but violates the architecture.