International Journal of Software Innovation
Volume 12 • Issue 1

The Study on Software Architecture Smell Refactoring

Kuo Jong-Yih, National Taipei University of Technology, Taiwan
Hsieh Ti-Feng, National Taipei University of Technology, Taiwan
Lin Yu-De, National Taipei University of Technology, Taiwan
Lin Hui-Chi, National Taipei University of Technology, Taiwan*
https://orcid.org/0000-0002-0492-5428

ABSTRACT

Maintenance and complexity issues in software development continue to increase because of new requirements and software evolution, and refactoring is required to help software adapt to the changes. The goal of refactoring is to fix smells in the system. Fixing architectural smells requires more effort than other smells because it is tangled in multiple components in the system. Architecture smells refer to commonly used architectural decisions that negatively impact system quality. They cause high software coupling, create complications when developing new requirements, and are hard to test and reuse. This paper presented a tool to analyze the causes of architectural smells such as cyclic dependency and unstable dependency and included a priority metric that could be used to optimize the smell with the most refactoring efforts and simulate the most cost-effective refactoring path sequence for a developer to follow. Using a real case scenario, a refactoring path was evaluated with real refactoring execution, and the validity of the path was verified.

KEYWORDS

Architecture Smell, Refactoring Strategies, Refactoring Tool

INTRODUCTION

In the software development life cycle (SDLC), the scale of a software project will grow because of the evolution in software requirements, IT equipment upgrades, and technology change (Lehman et al., 1996), which cause the cost of software maintenance and its complexity to increase. In order to maintain the quality of a project, teams will need to perform code refactoring regularly to reduce the accumulation of project technical debt (Suryanarayana et al., 2014). The best chance to do refactoring in a project is the region where smells are located. The smell is a surface indication that usually corresponds to a deeper problem in the system (Fowler et al., 1999). It can be classified into code smell (Fowler et al., 1999), design smell (Suryanarayana et al., 2014), and architectural smell (Lippert et al., 2006).

DOI: 10.4018/IJSI.339884

*Corresponding Author

Architecture smell (AS) is defined as common, but not always intentional, solutions used in architectural decisions that negatively impact software quality (Garcia et al., 2009). AS has relations with software architecture, and it may be involved in the division of a system into components, the arrangement of those components, and the ways in which those components communicate with each other (Martin, 2017).

The refactoring of AS involves coordinating a set of deliberate architectural activities that remove a particular architectural smell and improve at least one quality attribute without changing the system's scope and functionality (Sas et al., 2019). To help developers to remove AS, we developed a tool prototype as a support for AS refactoring that could analyze the actual cause of the AS and the recommended refactoring process based on the architecture smell using variable parameters and characteristic metrics (Arcelli et al., 2017).

The remainder of this paper is structured as follows: the second section introduces relevant terms in the field of architectural smells (AS), architectural smell refactoring, and related tools. The third section presents the research methodology used in this study and outlines the design of the refactoring process strategies. The fourth section describes the implementation of the device, presents a case study, and analyzes the results. Section five serves as the conclusion of the research.

RELATED WORK

Architecture Smell

Architecture smell is considered to violate the common design principle and affects the internal quality of software. It increases the coupling of components and may break the modularity of the system. Different authors have provided different definitions of AS according to different levels, such as Lippert et al., (2006), who defined AS's in dependency graphs, packages, subsystems, layers, and so on. Fontana et al., (2019) propose a tool called Arcan developed for the detection of architectural smells. Evaluate the PageRank and Criticality of these smells through the analysis of six projects These architectural smells are categorized into three types based on dependency issues, such as cyclic dependency (CD), unstable dependency (UD), and hub-like dependency (HL). This analysis has provided the architecture smell related to dependency issues, such as cyclic dependency (CD), unstable dependency (UD), and hub-like dependency (HL). Azadi et al., (2019) provide a proposal for AS classification (Figure 1) based on the violation of three design principles, including the principles of modularity (Suryanarayana et al., 2014), hierarchy (Suryanarayana et al., 2014), and healthy dependency structure (Caracciolo et al., 2016).

The AS chosen in this study included CD and UD, which can be detected by the Arcan tool in the detection of three smells in two industrial projects (Arcelli et al., 2016), and both violate the principle of the healthy dependency structure. CD also violates the principle of modularity, making it difficult to modify the requirements in the system and affecting the changeability and reusability of components related to the AS.

Cyclic Dependency

Cyclic Dependency (CD) represents a cycle among several components; it will lead the side effect when we try to modify the components in cycle. There are several software design principles that suggest avoiding creating such cycles, like Acyclic Dependencies Principle (Martin, 2003) and The Common Closure Principle (Robert, 2003). CD may have different topological shapes, which is shown in Figure 2, provided by Al-Mutawa et al., (2014). More complex shapes mean that the maintainability of the system is reduced because of the affected part.

Figure 1. Architectural Smell Classification

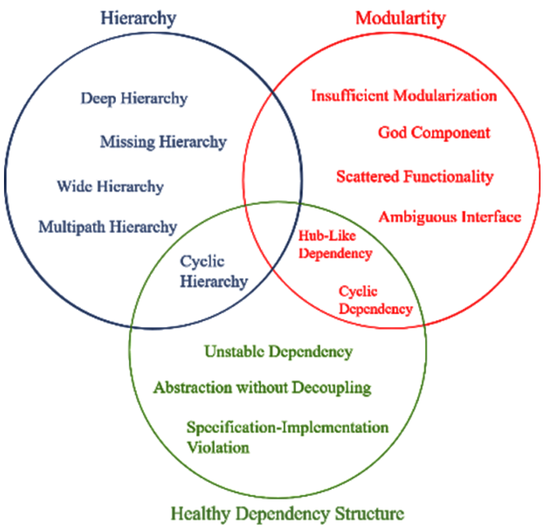
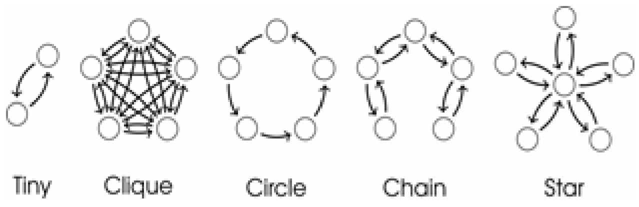


Figure 2. Type of Cycle Shapes



The Metrics of Refactoring the Candidate Priority

Developers use the metrics of refactoring the candidate priority to help decide on the best refactoring candidate to perform the corresponding refactoring strategies. Among the two types of smells in this study, there is more related research on the priority index of the cyclic dependence smell. Caracciolo et al., (2016) proposed calculating the number of dependencies between classes (class dependency), the number of dependencies between packages in a cycle (package dependency), and the number of shared package dependencies (shared package dependency) in a cycle in the dependency graph, as package dependencies severity index metric, shown as in Equation (1):

$$rank\left(pd\right) = \frac{\left| \{cycle \middle| pd \in SPD_{cycle} \right|}{CD_{pd}} \tag{1}$$

With the formula, we can consider the number of dependencies and the overlap degree of the dependencies as the smell priority metric. The cycle which has a smaller number of dependencies and a larger overlap degree with other cycles will have the highest priority. Rizzi et al., (2018) provides another formula for selecting the dependency in CD with the highest priority, as shown as in Equation (2):

$$\mathrm{P} = \frac{\mathrm{w0}}{cycles+1} + \frac{w1}{step} + w2\frac{\left(1-I_{from}\right)+\left(1-I_{to}\right)}{2}Ca_{from} + w3\frac{A_{from}+A_{to}}{2} \tag{2}$$

Rizzi's formula extends the priority metric of the study by the priority metric of the dependency, considering the Ca_{from}, $I_{from/to}$, and $A_{from/to}$. However, the P value of Equation (2) will be greatly affected because of the Ca_{from} metric.

Based on the above, this research combined the metrics of Rizzi et al., (2018) and Caracciolo et al., (2016) and followed the stable-dependencies principle. A component should only depend on components that are more stable than itself.

The Refactoring Technology for Architecture Smell

Architectural smell refactoring is related to architectural refactoring, which is identified defined as a series of prudent architectural behaviors according to Zimmermann (Sas et al., 2019). With the goal of eliminating architectural smells, it can be extended by combining different types of techniques based on the selected abstraction level of the analysis project information. According to Baqais et al., (2020), among 41 selected papers related to refactoring technology, the selected abstraction levels include the use of unified modeling language (UML) (2022), models, graphs, and package levels. It also included the use of the following four categories of techniques, first including search-based algorithms to get the best solution, second similarity-based algorithms on the similarities between the test sample and a set of labeled training samples, third agglomerative Hierarchical cluster algorithm as a form of bottom-up clustering, finally, metric-based algorithms to estimate the robot planar displacement by matching dense two-dimensional range. Praditwong et al., (2011), approaches to modularization focus on automated algorithms that seek out new partitions of software, aiming to maximize cohesion and minimize coupling. It improves the measurements of cohesion and coupling that can be achieved so that, where this is desirable, the user will have potentially more interesting suggestions from the tool to consider.

For example, Bavota et al., (2014) used the similarity matrix between functions to find out the candidates for the extraction using the class method. Clustering algorithms are used to classify the component data in the project into groups, like in Chantian et al., (2019), to propose a refactoring approach for Too Large Packages smell, which is one of the architecture smells by using community detection in extracting process. Metric-based algorithms are based on measuring software metrics and are used to find candidates for refactoring by comparing different versions of the project and comparing the degree of index change.

In this study, a method that tries to define the static dependency graph of the project is proposed, the priority formula composed of the measurement index of the component and the characteristics of each architectural smell is proposed, reconstruction steps of the improved architectural smell are proposed, and the candidates for refactoring are obtained by measuring the index to solve the architectural smell.

THE PROPOSED APPROACH

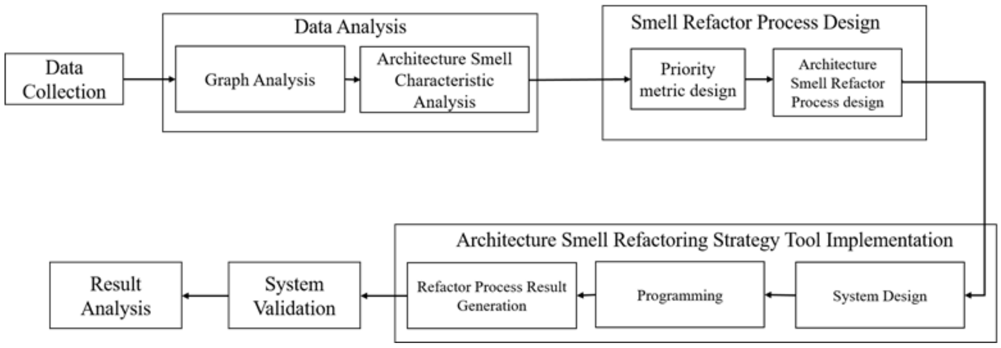
Research Processing Diagram

The research process of this paper is shown in Figure 3.

Data Collection

This research used the dependency graph data provided by Arcan, an AS detection tool that can analyze project source codes by setting the project path. The graph data contained the metrics of components, such as packages or classes, as attributes of the nodes in the graph.

Figure 3. The Research Process of This Paper


Data Collection
Data Analysis
Graph Analysis
Architecture Smell Characteristic Analysis
Smell Refactor Process Design
Priority metric design
Architecture Smell Refactor Process design
Architecture Smell Refactoring Strategy Tool Implementation
System Design
Programming
Refactor Process Result Generation
System Validation
Result Analysis

Data Analysis

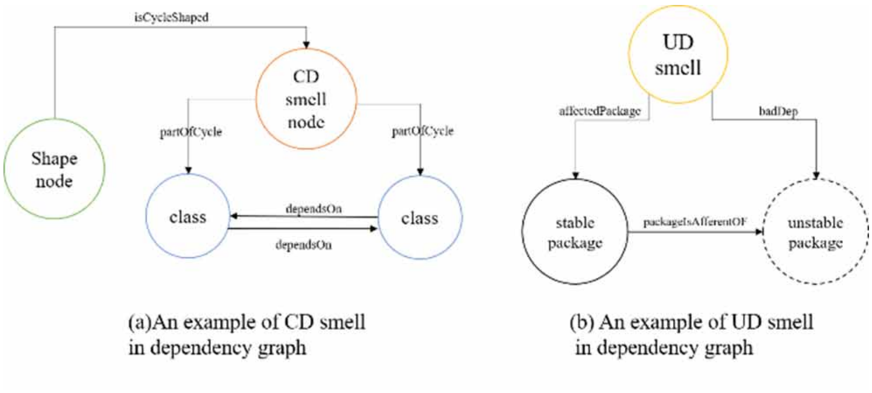
Dependency graph data analysis: The dependency graph data generated by Arcan contained node elements and edge elements. The node types included component nodes, smell nodes, and shape nodes. The component node included package nodes and class nodes to represent the package and class in the Java project. The smell node was used to mark AS, such as CD, UD, and HL, and would point to the set of component nodes related to the AS, which could be regarded as a corresponding set for the range of nodes affected by AS. The shape node was used with CD to mark its shape CD constituted.

This study analyzed the component nodes related to the architecture smell using the edge relationships among nodes. The relationship between CD and UD in the dependency graph is shown in Figure 4.

Architecture Smell Characteristics Analysis

An architectural smell characteristic is a property or attribute of an architectural smell (Sas et al., 2019). Architectural smell characteristics can be used to help quantify the severity of the degree to which an AS affects the system. Architectural smell characteristics can be classified into generic smell characteristics, which can be measured for every type of smell (such as the size and number of edges in AS), and specific smell characteristics that can only be measured for certain types of smells, such as shared package dependencies (SPD) and the edge cycle sharing degree (ECSD) (Caracciolo et al., 2016).

Figure 4. The Relation Between CD and UD Dependency Description in a Graph



Architecture Smell Refactoring Priority Index

This research design first selected the CD with the highest smell refactoring priority index as the basis for the CD refactoring process and then processes the priority of dependencies that constitute odors. For the smell refactoring priority index of CD, this research does the formula adjustment; the formula provided by Caracciolo et al., (2016) suggests this is the most cost-effective sequence of refactoring operations that will break the cycle. The adjusted formula was as shown in Equation (3):

$$rank\left(CD\right) = \frac{|\ SPD_{CD}\ | + c}{\left|DIP_{CD}\right| * |\ DIC_{CD}\ |} \tag{3}$$

We could find the edge refactoring priority index of the dependency which is the most recommended. We also provided the adjusted formula, as shown in Equation (4):

$$P'_{CD}\left(Edge\right) = w0\left(1 - I_{from}\right) + w1\left(A_{from}\right) + w2 * ECSD_{Edge} + w3 * \frac{1}{EIR_{Edge}} \tag{4}$$

In the refactoring process for UD, this research first selected the UD with the highest smell refactoring priority index and selected the dependency with the highest edge refactoring priority index to constitute the UD. Fontana et al., (2019) provided the metric reference packages (RP) as the criticality of UD and suggested the criticality of the smell can be a consideration for refactoring. The smell refactoring priority index for UD was as shown in Equation (5):

$$rank\left(UD\right) = \left|RP\right| \tag{5}$$

As a UD may consist of more than one unstable relationship, we used the other metric of UD—the instability gap, as the edge refactoring priority index to help to select the most unstable relationship, as shown in Equation (6):

$$P'_{UD}\left(Edge\right) = I_{to} - I_{from} \tag{6}$$

After selecting the relationship with the highest P'_{UD}, we further analyzed the actual cause of the relationship to find the class dependencies leading the package relationship and suggest the best refactoring process for the class dependencies, as shown in Equation (7):

$$P'_{UD}\left(EdgeInClasses\right) = w0\left(1 - I_{from}\right) + w1\left(A_{from}\right) \tag{7}$$

Architecture Smell Refactoring Strategies

Refactoring Strategies for CD

Regarding the refactoring strategies for CD, we adjusted the refactoring process provided by the Rizzi et al., (2018) process of explaining how the dependencies between the suites are composed and how to refactor. We analyzed the total CDs in the project, calculated the rank, found the CD with the highest rank, and analyzed the edges constructing the CD to find the edge with the highest P'_{CD}. If the edge represented a package dependency, we would analyze the actual cause of the package dependency to provide more information about the CD.

The adjusted CD reconstruction process in this study is shown in Figure 5.

Refactoring Strategies for UD

Regarding the refactoring strategies used for UD, we analyzed the total UDs in the project shown in Figure 6, and sorted all UDs in the system to find out the UDs that should be processed first in the system. Based on this UD, we found out the unstable relationship that should be dealt with first and analyze the actual cause of the unstable relationship, and found the edge with the highest P'_{UD} . As the unstable relationships all belonged to a package dependency, we directly analyzed the actual cause of the package dependency.

PRACTICE AND EXPERIMENT

This section introduces the implementation method for the architecture smell refactoring support tool. In order to evaluate the tool, we used a proof-of-concept prototype to analyze one open-source project and invited subjects to participate in a refactoring experiment for the evaluation of our tool.

Figure 5. Path Model of Relations Among Study Variables

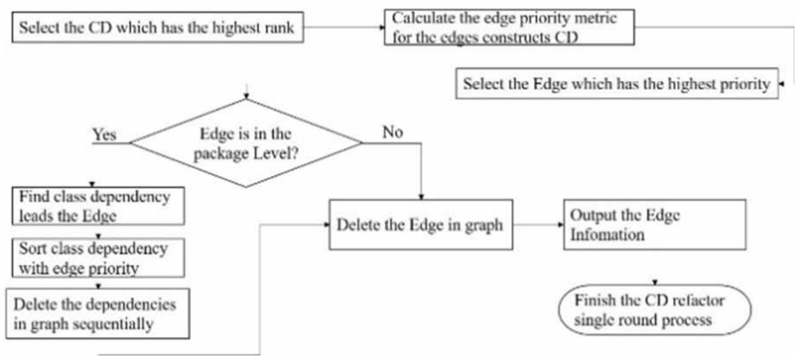
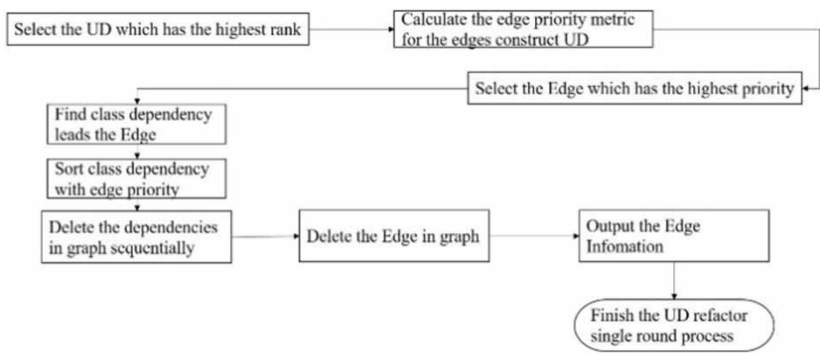

Select the CD which has the highest rank
Calculate the edge priority metric for the edges constructs CD
Select the Edge which has the highest priority
Yes
Edge is in the package Level?
No
Find class dependency leads the Edge
Sort class dependency with edge priority
Delete the dependencies in graph sequentially
Delete the Edge in graph
Output the Edge Infomation
Finish the CD refactor single round process

Figure 6. The UD Refactoring Process Design in This Research


Select the UD which has the highest rank
Calculate the edge priority metric for the edges construct UD
Select the Edge which has the highest priority
Find class dependency leads the Edge
Sort class dependency with edge priority
Delete the dependencies in graph sequentially
Delete the Edge in graph
Output the Edge Infomation
Finish the UD refactor single round process

System Flow Chart

The flow chart of the architecture smell refactoring support tool proposed in this research is shown in Figure 7.

Experiment I: Informa Project Refactoring

Case Information

We used the Informa used by Rizzi et al., (2018) as a case for experimental comparison as the case for architecture smell refactoring. Informa provides a Java-based library for RSS (simple syndication) access. It supports data extraction from channels (such as websites) in different formats and notifies the user whenever information is updated. The basic information of Informa Projects' metrics is shown in Table 1.

The architecture of Informa is shown in Figure 8. There are six packages in Informa project architecture. The parser package is used to provide the RSS data parser in a different format. The core package is used to provide the interface class of the domain objects and then implement it by the class in the impl package. The impl package provides two ways to implement a core object, implementation methods storing data in memory (basic) and using an external database framework (hibernate). The utils package contains the manager package, which manages the channel and provides RSS data, the cleaner package, which is used to clean data in the channel, the poller package, which is used to manage user notifications, and the toolkit package, which performs task scheduling. The search package provides the search function for RSS data, and the exporter's package provides the RSS output data in a different format.

This study used Arcan version 1.2.1 as the research tool. After analyzing Informa, the total number of architecture smells detected was as shown in Table 2. The most common architectural smell in the project was CD.

Figure 7. Flow Chart of the Architecture Smell Refactoring Support Tool

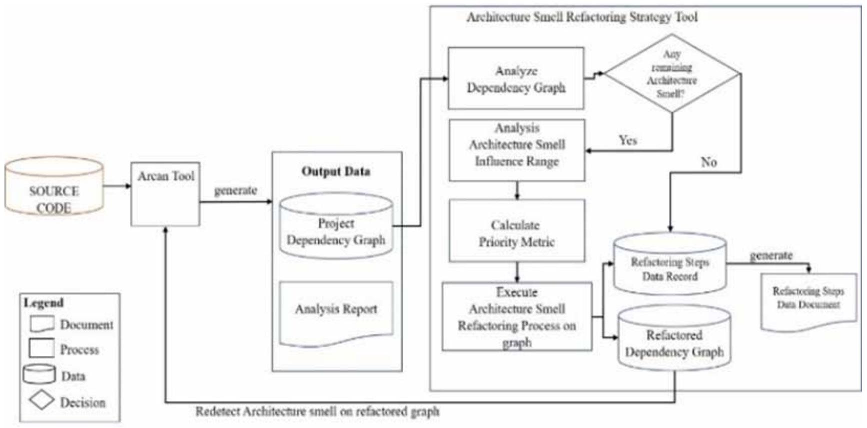

Architecture Smell Refactoring Strategy Tool
SOURCE CODE
Arcan Tool
generate
Output Data
Project Dependency Graph
Analysis Report
Analyze Dependency Graph
Any remaining Architecture Smell?
Yes
No
Analysis Architecture Smell Influence Range
Calculate Priority Metric
Execute Architecture Smell Refactoring Process on graph
Refactoring Steps Data Record
generate
Refactoring Steps Data Document
Refactored Dependency Graph
Redetect Architecture smell on refactored graph
Legend
Document
Process
Data
Decision

Table 1. The Informa Metrics

Project Name	Version	NOP	NOC	LOC
Informa	0.7.0	19	199	9722

Figure 8. The Architecture of Informa

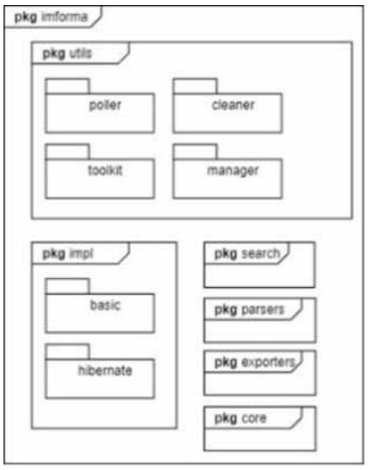


Table 2. Number of Architecture Smells in Informa

Project Name	Numbers of CD	Numbers of UD
Informa	18	3

Result for Refactoring CD

CD can be categorized into the class level and package level. The corresponding CDs in Informa are shown in Table 3.

After analyzing the CDs in Informa, we found many different causes of CD at the package and class levels, as shown in Figure 9. There were two reasons for CD at the class level in Informa: one was the dependency between classes, as shown in Figure 9(c), and the other was the dependency between classes and its inner classes, as shown in Figure 9(d). The reason for CD at the package level in Informa was the dependency between packages, like that shown in Figure 9(a), and the overlap of complex cycles with other CDs, like that shown in Figure 9(b).

Following the steps as mentioned in Chapter 3, Section 5, after analyzing the CD, based on Chapter 3, Section 4 (Equation 3), we can calculate the smell refactoring priority index of CD in Informa, so that the tool will select the CD utils $\Leftrightarrow$ impl.basic who has the highest rank(0.175) as the next refactoring candidate. After finding the CD, the tool will further analyze the dependencies that constitute the CD, and calculate the edge refactoring priority index of the dependencies, as shown in Table 5. That tool will select the edge *impl.basic-packageIsAfferentOf*->utils which has the highest P as the refactoring edge candidate. If the chosen edge's level is package, the tool will analyze the class dependencies that lead the chosen edge. For the edge *impl.basic-packageIsAfferentOf*->utils, we can find the class dependencies shown in Table 4.

Table 3. Number of CDs at Different Levels in Informa

Numbers of CD	CD in package level	CD in class level
18	5	13

Figure 9. The Cause of CD in Informa

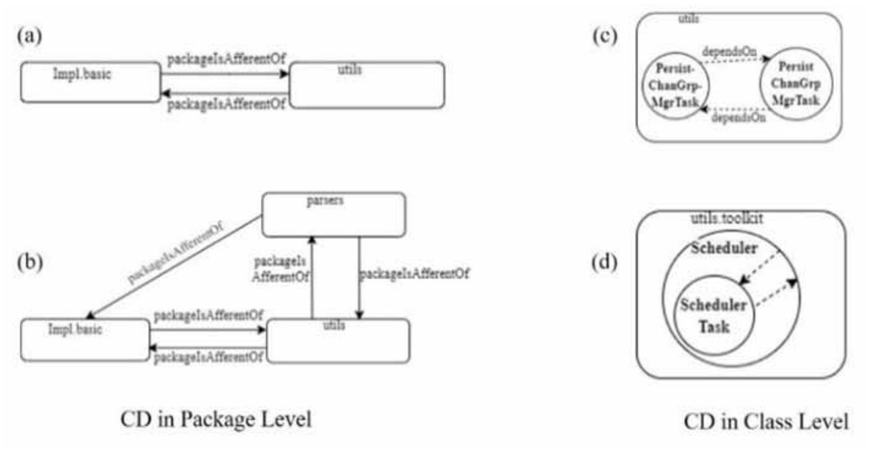


Table 4. Dependencies Metric of Selected CD

Edge	P	Level
utils-packageIsAfferentOf->impl.basic	2.8145	package
impl.basic-packageIsAfferentOf->utils	2.9137	package

Table 5 shows the total refactoring sequences for eliminating all CDs in Informa generated by the tool. The total 18 CDs will be eliminated in 17 steps with three suggested refactoring sequences.

Comparison With Rizzi

In order to make a comparison with the Rizzi et al., (2018) refactoring steps and the proposed refactoring steps, this research implemented Rizzi's architectural smell refactoring strategy. The biggest difference between the proposed strategy and Rizzi's refactoring strategy was that the proposed strategy could select the smell with the highest rank first, as it would be the most cost-effective smell, would have fewer dependencies, and would have the highest degree of overlap with other cycles in the project. The comparison between the proposed strategy and that of Rizzi is shown in Table 6.

Result for Refactoring UD

Three UDs were detected in Informa, and the smell distribution was shown in Figure 10.

Based on Equation (5), the Rizzi Refactoring Path selected the UD with the most RPs; therefore, the UD related to utils -> impl.hibernate and utils -> parsers were selected first. Because the UD had two unstable dependencies, the tool calculated the instability gap between the dependencies, selected the dependency with the highest instability gap, and analyzed the class dependencies constituting the package dependency. After analyzing the UD, the tool deleted the related dependencies and repeated the process until all UDs were deleted.

Experiment II: Refactoring With Subjects

In order to evaluate the information generated by the tool, we invited six subjects to analyze software projects and refactor and diagnose architectural smells with and without the architectural smell

Table 5. Refactoring Sequences for Eliminating All CD In Informa

Steps	Refactoring Sequences Generated by Tool
1	impl.basic-**packageIsAfferentOf**-> utils
	impl.basic.**Item-dependsOn**-> utils.XmlPathUtils
	impl.basic.Channel-**dependsOn**-> utils.XmlPathUtils
2	utils.PersistChanGrpMgr-**dependsOn**-> utils.PersistChanGrpMgrTask
3	utils.poller.Poller$SchedulerCallback-**dependsOn**-> utils.poller.Poller
4	parsers.RSS_1_0_Parser$RSS_1_0_ParserHolder-**dependsOn**-> parsers.RSS_1_0_Parser
5	parsers.Atom_1_0_Parser$Atom_1_0_ParserHolder-**dependsOn**-> parsers.Atom_1_0_Parser
6	utils.cleaner.Cleaner$SchedulerCallback-**dependsOn**-> utils.cleaner.Cleaner
7	parsers.RSS_2_0_Parser$RSS_2_0_ParserHolder-**dependsOn**-> parsers.RSS_2_0_Parser
8	utils.cleaner.Cleaner$CleanerThreadFactory-**dependsOn**->utils.cleaner.Cleaner
9	parsers.RSS_0_91_Parser$RSS_0_91_ParserHolder-**dependsOn**-> parsers.RSS_0_91_Parser
10	utils.poller.Poller$PollerThreadFactory-**dependsOn**->utils.poller.Poller
11	utils.FeedRefreshDaemon-**dependsOn**-> utils.FeedRefreshDaemon$FeedRefreshTask
12	utils.ChannelRegistry-**dependsOn**-> utils.UpdateChannelTask
13	utils.toolkit.Scheduler-**dependsOn**-> utils.toolkit.Scheduler$SchedulerTask
14	parsers.Atom_0_3_Parser$Atom_0_3_ParserHolder-**dependsOn**-> parsers.Atom_0_3_Parser
15	utils.toolkit-**packageIsAfferentOf**-> utils.poller
	utils.toolkit.WorkersManager-**dependsOn**->utils.poller.PriorityComparator
16	utils-**packageIsAfferentOf**-> impl.hibernate
	utils.PersistChanGrpMgr-**dependsOn**-> impl.hibernate.Channel
	utils.PersistChanGrpMgr-**dependsOn**-> impl.hibernate.ChannelBuilder
	utils.PersistChanGrpMgr-**dependsOn**-> impl.hibernate.ChannelGroup
	utils.PersistChanGrpMgrTask-**dependsOn**-> impl.hibernate.ChannelBuilder
	utils.PersistChanGrpMgrTask-**dependsOn**-> impl.hibernate.Channel
	utils.PersistChanGrpMgrTask-**dependsOn**-> impl.hibernate.ChannelGroup
17	utils-**packageIsAfferentOf**-> parsers
	utils.FeedManagerEntry-**dependsOn**-> parsers.FeedParser
	utils.UpdateChannelTask-**dependsOn**-> parsers.FeedParser
	utils.PersistChanGrpMgrTask-**dependsOn**-> parsers.FeedParser
	utils.FeedManager-**dependsOn**-> parsers.OPMLParser
The numbers of total sequences	30

refactoring in Informa and with and without the architecture smell refactoring support tools, including Arcan, Rizzi's method, and the proposed tool in this research.

All the subjects were first-year Master of Computer Science and Information Engineering students at NTUT (National Taipei University of Technology), and all of them had taken software engineering and object-oriented analysis and design lessons. The record of the course scores is shown in Table 7, and the scores are in the range of 81 to 89.

There were seven stages in this experimental design. The first stages tested the subjects' domain knowledge with a comprehension quiz, which was used to make sure that all the subjects had a

Table 6. Comparison with Refactoring Steps in this Study with Rizzi Refactoring Steps

Project Name	This Research	Rizzi's
Rounds	17	17
Total Refactor Sequences	30	17
Considering the edge overlap degree	Yes	No
Analyze the real reason led to the edge.	Yes	No
Provide detailed information on CDs	Yes	No

Figure 10. Total UDs in Informa

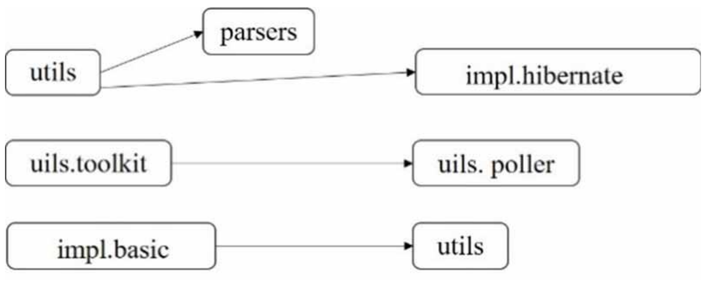


Table 7. Records of Subjects Taking Courses

Subject ID	Software Engineering	Object-Oriented Analysis and Design
A	88	84
B	84	84
C	81	86
D	83	86
E	86	86
F	85	86

unified understanding of the experimental related fields. The authors designed a pretest quiz and a post-test quiz. In the second stage, we investigated if the subjects have a consistent understanding of the domain after the education and training.

Stage four was the introduction to the experiment and the opening for Informa, including the software architecture of Informa, the definition of RSS, and how to use Informa to develop a simple RSS reader. The introduction to the experiment defined AS, explained how to perform refactoring, and let the subjects know how to identify the region of the AS, analyze the actual cause, and perform the refactoring strategies.

Stages five and six were the architecture smell refactoring diagnosis experiments without and with the architecture smell refactoring support tool, every stage remained 90 minutes. The architecture smells refactoring support tools included Arcan, Rizzi's method, and the proposed research tool. To make sure the subjects would not know which tool was being used, the tools were named tool A (Arcan), tool B (Rizzi's method), and tool C (the proposed method).

The questionnaire in this study was designed according to a Likert scale (1932) and included eight questions. Questions 1–3 measured the subjects' feelings about the first part of the experiment without any architecture smell refactoring support tool. Question 4 was used to ask about the architecture smell support tools used by the subjects. Questions 5–7 measured the subjects' feelings about the second part of the experiment with the architecture smell refactoring support tool. Question 8 measured the subjects' feelings about the tools they used and if it helps to save time during the whole refactoring process. The questionnaire used a seven-point scale for scoring (except for question 4), with a score of 1 to 7 indicating strongly agree, agree, somewhat agree, neither agree nor disagree, somewhat disagree, disagree, and strongly disagree, respectively.

The Result of the Experiment

The pretest and posttest results are shown in Table 8. The results indicated that after the experiment-related domain knowledge education and training stage, all the scores were higher.

The box plot of Table 8 is shown in Fig. 11. The results indicated that the posttest IQR was better than the pretest IQR, meaning the subjects' understanding of the domain had improved, while the understanding criteria of the subjects tended to be at the same level.

Table 9 presents a record of each process of the architecture smell refactoring diagnosis from the first part of the experiment. To carry out the architecture smell refactoring diagnosis without using a tool, the subjects first needed to compare the dependencies between classes or packages in Informa according to the definition of CD smell. This required the subjects to perform manual comparisons of many source codes to determine the circular dependency odor.

Table 8. Records of Subjects Taking Courses

Subject ID	Pretest score	Posttest score
a	88	100
b	72	96
c	72	92
d	84	96
e	76	100
f	80	100
	Pretest IQR	Posttest IQR
	10	4

Figure 11. Box Plot of the Subjects' Domain Knowledge Comprehension Quiz

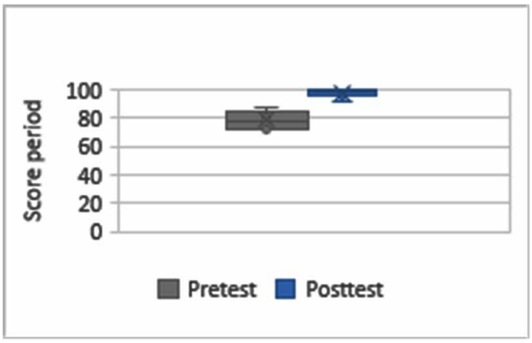


Table 9. The Number of CDs Finished in Each Process in Part One

Subject ID	The numbers of found CDs in part one.	The numbers of the CDs which the actual cause analysis finished in part one.	The numbers of the CDs which the refactoring strategies analysis finished in part one.	The numbers of the CDs which the whole refactoring diagnosis analysis finished in part one.
a	0	0	0	0
b	3	3	3	3
c	3	3	2	2
d	1	1	0	0
e	0	0	0	0

Table 10 is the record of each process of the architecture smell refactoring diagnosis from the second part of the experiment, in which subject a and subject d use A tool - Arcan tool; subject b and subject e use B tool – Rizzi's tool; subject c and subject f use the C tool - a tool developed for this research.

We observed that the number of CDs finished at each stage of the refactoring process by all subjects was higher. This indicates that the information provided by these three support tools helped the subjects to perform the refactoring process for architecture smell and helped to increase the number of smells that complete the refactoring diagnosis.

For the comparison of individual tools, the total number of CDs that the tool completed the whole smell refactoring diagnosis for and the difference between part one and part two of the CDs, which the whole refactoring diagnosis analysis finished with the tool, is shown in Table 11. We observed that tool C (the proposed tool in this study) had the highest number in both fields.

CONCLUSION

This research provided a prototype of an architecture smell refactoring support tool with the dependency graph analyzed by Arcan. The tool could analyze the cyclic dependency and unstable dependency smell, including the actual cause of the smell and the recommended refactoring process for eliminating all smells in a project, implemented by refactoring strategy for CD and UD with the combination of the architecture smell characteristics.

Table 10. The Number of CDs Finished in Each Process in Part Two

Tool ID	Subject ID	The numbers of found CDs in part two.	The numbers of the CDs which the actual cause analysis finished in part two.	The numbers of the CDs which the refactoring strategies analysis finished in part two.	The numbers of the CDs which the whole refactoring diagnosis analysis finished in part two.
A	a	4	4	4	4
A	d	5	5	5	5
B	b	6	6	6	6
B	e	5	5	5	5
C	c	6	6	6	6
C	f	7	7	7	7

Table 11. Comparison Between Tools

Tool	The total number of CDs which the whole refactoring diagnosis analysis finished with the tool.	The difference between part one and part two of the CDs which the whole refactoring diagnosis analysis finished with the tool.
A	9	9
B	12	8
C	13	10

The results of Experiment I show that the information provided by the tool was more detailed than that provided by Rizzi's method. The results of Experiment II show that the proposed tool had a better score than that provided by Arcan or Rizzi's tool. Therefore, the tool could help developers save the time needed to analyze architecture smells and the time needed to calculate the metrics of the architecture smell characteristics.

In the future, we hope our tool will be able to provide more architecture smell refactoring support for different smells, such as Hub-like dependencies or God -component smells and add the suggestion for the refactoring method, like extract method or move class, etc. The tool's recommended refactoring process can become closer to the refactoring process of actual developers and provide more information on how to refactor the smell.

In the future, we hope to invite different types of subjects, such as junior and senior developers, to clarify what different types of developers think about the architecture smell refactoring support tool, and keep adjusting the function and information of the tool based on feedback, to let the users improve the quality of the software project.

CONFLICTS OF INTEREST

There are no conflicts of interest among authors.

FUNDING

This research was supported by the National Science Council under grant number MOST 111-2221-E-027-133.

REFERENCES

Al-Mutawa, H. A., Dietrich, J., Marsland, S., & McCartin, C. (2014). On the shape of circular dependencies in Java Programs. *Proceedings of the 2014 23rd Australian Software Engineering Conference (ASWEC '14)*, 48–57. doi:10.1109/ASWEC.2014.15

Arcelli, F. F., Pigazzini, I., Roveda, R., & Zanoni, M. (2017). Automatic detection of instability architectural smells, Automatic Detection of Instability Architectural Smells. *Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution*, 433-437. doi:10.1109/ICSME.2016.33

Azadi, U., Arcelli, F. F., & Taibi, D. (2019). Architectural smells detected by tools: a A catalogue proposal. *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 88-97. doi:10.1109/TechDebt.2019.00027

Baqais, A. A. B., & Alshayeb, M. (2020). Automatic software refactoring: A A systematic literature review. *Software Quality Journal*, *28*(2), 459–502. doi:10.1007/s11219-019-09477-y

Bavota, G., De Lucia, A., Marcus, A., & Oliveto, R. (2014). Automating extract class refactoring: An An improved method and its evaluation. *Empirical Software Engineering*, *19*(6), 1617–1664. doi:10.1007/s10664-013-9256-x

Caracciolo, A., Bledar, A., Lungu, M., & Nierstrasz, O. (2016). Marea: A semi-automatic decision support system for breaking dependency cycles. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER),* 482-492. doi:10.1109/SANER.2016.11

Chantian, B., & Muenchaisri, P. (2019). A refactoring approach for too large packages using community detection and dependency-based impacts. In *Proceedings of the 1st World Symposium on Software Engineering (WSSE '19).* Association for Computing Machinery. doi:10.1145/3362125.3362132

Fontana, F. A., Pigazzini, I., Raibulet, C., Basciano, S., & Roveda, R. (2019). Pagerank and criticality of architectural smells. *Proceedings of the 13th European Conference on Software Architecture*, *2*, 197-204. doi:10.1145/3344948.3344982

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the design of existing code* (1st ed.). Addison-Wesley Professional.

Garcia, J., Popescu, D., Edwards, G., & Medvidovic, N. (2009). Identifying architectural bad smells, *2009 13th European Conference on Software Maintenance and Reengineering*, 255-258. doi:10.1109/CSMR.2009.59

Lehman, M. M. (1996). Laws of software evolution revisited. *Proceedings of the 5th European Workshop on Software Process Technology*, 108–124. doi:10.1007/BFb0017737

Likert, R. (1932). A technique for the measurement of attitudes. *Archives de Psychologie*.

Lippert, M., & Roock, S. (2006). *Refactoring in large software projects: performing Performing complex restructurings successfully*. John Wiley & Sons.

Martin, C. (2017). *Clean architecture: A craftsman's guide to software structure and design*. Prentice Hall Press.

Martin, R. (2003). *Agile software development: Principles, patterns, and practices*. Prentice Hall PTR.

Pan, W. F., Jiang, B., & Li, B. (2013). Refactoring software packages via community detection in complex software networks. *International Journal of Automation and Computing*, *10*(2), 27–31. doi:10.1007/s11633-013-0708-y

Praditwong, K., Harman, M., & Yao, X. (2011). Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, *37*(2), 264–282. doi:10.1109/TSE.2010.26

Rizzi, L., Fontana, F. A., & Roveda, R. (2018). Support for architectural smell refactoring. In *Proceedings of the 2nd International Workshop on Refactoring (IWoR 2018).* Association for Computing Machinery. doi:10.1145/3242163.3242165

Sas, D., Avgeriou, P., & Arcelli, F. F. (2019). Investigating instability architectural smells evolution: An exploratory case study. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 557–567. doi:10.1109/ICSME.2019.00090

Suryanarayana, G., Samarthyam, G., & Sharma, T. (2014). *Refactoring for software design smells: Managing technical debt. Morgan Kaufmann* (1st ed.). http://www.omg.org

Zimmermann, O. (2015). Architectural refactoring: A task-centric view on software evolution. *IEEE Software*, *32*(2), 26–29. doi:10.1109/MS.2015.37

Jong-Yih Kuo is currently a Professor in the Department of Computer Science and Information Engineering at Taipei University of Technology in Taiwan. He received a Ph.D. degree in the Department of Computer Science and Information Engineering from National Central University in Taiwan in 1998. His current research interests include software engineering and intelligent systems.

Ti-Feng Hsieh received the M.S. degree in Computer Science and Information Engineering from Taipei University of Technology, Taiwan. His research interests include software engineering, and machine learning.

Yu-De Lin received the M.S. degree in Computer Science and Information Engineering from Taipei University of Technology, Taiwan. Her research interests include software engineering, and machine learning.

Hui-Chi Lin is currently pursuing a master's degree in the School of Computer Science and Information Engineering at Taipei University of Technology in Taiwan. Her research interests include software engineering, and machine learning.