Measuring the Effects of Aspect-Oriented Refactoring on Component Relationships: Two Case Studies

Reishi Yokomori§, Harvey Siy†, Norihiro Yoshida‡, Masami Noro§, Katsuro Inoue*
§Department of Software Engineering, Nanzan University, Japan
†Department of Computer Science, University of Nebraska at Omaha, USA
‡Graduate School of Information Science, Nara Institute of Science and Technology, Japan
*Graduate School of Information Science and Technology, Osaka University, Japan
yokomori@nanzan-u.ac.jp, hsiy@mail.unomaha.edu, yoshida@is.naist.jp,
yoshie@nanzan-u.ac.jp, inoue@ist.osaka-u.ac.jp

ABSTRACT

Aspect-oriented refactoring is a promising technique for improving modularity and reducing complexity of existing software systems through encapsulating crosscutting concerns. As complexity of a system is often linked to the degree to which its components (e.g., classes and aspects) are connected, we investigate in this paper the impact of such refactoring activities on component relationships. We analyze two aspect-refactoring projects to determine circumstances when such activities are effective at reducing component relationships and when they are not. We measure two kinds of relationships between components, use and clone relations. We compare how these metrics changed between the original and the refactored system. Our findings indicate that aspect-oriented refactoring is successful in improving the modularity and complexity of the base code. However, we obtain mixed results when aspects are accounted for. Based on these results, we also discuss constraints to the technology as well as other design considerations that may limit the effectiveness of aspect-oriented refactoring on actual systems.

Categories and Subject Descriptors: D.2.8 [Software Engineering]:Metrics - Product metrics

General Terms: Measurement, Languages, Experimentation

Keywords: Use-relation analysis, code clone analysis, aspect-oriented programming, refactoring, coupling

1. INTRODUCTION

Many projects have adopted an incremental approach to software development. In such approach, readability and maintainability can deteriorate due to the accumulation of features, so developers perform refactoring activities to reorganize code structure and to prepare for future extensions. Refactoring[17] is a suite of activities for changing software's internal structure without modifying its existing functionality, in order to improve internal quality of the software, such as readability and complexity of source code. Refactoring has become one of the essential activities in development of large software systems, and there are a lot of studies, suggestions, and practices for different refactoring approaches.

We focus attention on a refactoring approach based on aspectization. A merit of aspect-oriented development[10] is the ability to modularize crosscutting code, that is, features implemented in a crosscutting manner would be separated from the structure of the base software source code and moved into their own modules, known as aspects. On the basis of these characteristics, extracting several features as aspects from the existing system is suggested as a refactoring approach.

A commonality across most refactoring techniques is that they simplify code by consolidating similar pieces of code. We expect this effect to be even more pronounced with aspect-oriented refactoring as it enables consolidation of crosscutting code in ways that would not be possible with conventional refactoring techniques[13].

In this paper, we undertake an empirical study to assess how this consolidation affects existing software systems. Specifically, we investigate the following research questions:

Q1 Is aspect-oriented refactoring effective for improving modularity and complexity?

Q2 What are the characteristics of classes likely to be strongly affected by such refactoring activities?

We address these questions with respect to the effects on component relationships. The degree of relatedness of a system's components affects its modularity, which in turn affects the complexity of maintaining it.

1.1 Overview of Methodology

We employ a multiple-case study approach to investigate these questions. We identify several real-world projects that have aspectized counterparts with the same functionalities. We compare the differences in the aspectized and original versions in terms of two classes of metrics: change in use relations and change in clone relations.

Use relations indicate coupling between components in terms of usage dependencies. We compute use relations as a surrogate measure for modularity. More modular systems are expected to have fewer couplings. We conjecture that crosscutting code that are part of the same concern are somehow interrelated in the sense that they have usage dependencies with each other or use some other common component. By moving crosscutting code into aspects, scattered usage dependencies between crosscutting code will now be localized within the aspect. Also, usage dependencies with common components will be consolidated and will originate from the aspect rather than from multiple sources.

Clone relations indicate common code fragments shared by two or more components. Code for many common crosscutting concerns such as logging and tracing are often homogeneous, that is, they share similar or nearly identical code fragments [4]. The presence of such clones increases the complexity of subsequent maintenance activities due to the potential need to update multiple places to implement a change. By refactoring them into aspects, homogeneous crosscutting code will be factored out of the base and into aspects, thus reducing complexity of maintenance. Clone relations provide a complementary perspective for tracking the refactoring process. While tracing use relations enables us to trace the movement of programmatic relationships, tracing clone relations enables us to trace the actual movement of code from base to aspects.

We refine our research questions in terms of these metrics:

Q1-1 Is refactoring effective for reducing use and clone relations between classes?

Q1-2 Is refactoring effective for reducing total use and clone relations (classes and aspects)?

Q2-1 What kinds of components are likely to have fewer use relations after refactoring?

Q2-2 What kinds of components are likely to have fewer clone relations after refactoring?

We analyze two large software applications that have been refactored into aspects, JHotDraw and Berkeley DB. In both projects, Java programs were refactored and some functions were extracted and rewritten as aspects using AspectJ. For each project, we compare the complexity of original structure with the refactored structure, first without the aspects, and then with aspects taken into account. By tracking the changes to use and clone relationships, we can characterize refactoring activities in a way that enables us to reach some conclusions about the effectiveness of aspect-oriented refactoring as well as explain the observations recorded by the original refactoring teams.

In Section 2, we introduce two types of component graphs, which provide the conceptual model for the analysis of use and clone relations. The results of analyzing two refactoring projects are presented in Section 3. We discuss the effectiveness of the refactoring by using aspects and related works in Section 4.

2. BACKGROUND

2.1 Software Component

In general, a component is a modular part of a system that encapsulates its content and whose manifestation is replaceable within its environment [11]. It represents a logical unit in the program source code, such as a class, function, or package, etc. In this study, we treat classes and aspects as components. We model relationships between components using component graphs. We introduce two types of component graphs, one based on use relations and another based on clone relations.

2.2 Use Relation Component Graph

The component graph based on use relations is modeled as a directed graph. A node in the graph represents a software component, and a directed edge from node x to y represents a *use relation* meaning that component x uses component y. By using the graph, we can easily identify the use relations between components and count the incoming and outgoing edges of a component.

Use relation metrics are calculated using SPARS-J[7]. SPARS-J is a web-based Java code search engine and navigates a lot of registered components based on use relation between components. We use it to compute the use relations. SPARS-J identifies use relations using static analysis, so dynamic binding is excluded. In the graph, we treat Java classes as components, and consider the following as use relations:

- inheritance,
- implementation of abstract class and interface,
- declaration of variables,
- creation of instances,
- method calls, and
- class attribute references.

In some analyses involving aspectized versions, we also treat aspects as components, and take into consideration use relations between aspects and classes to understand how code fragments in original classes spread to aspects. We define two kinds of use relation; those are use relations from classes to aspects and the ones from aspects to classes.

A directed edge from aspect a to class c represents a use relation meaning that at least one advice from a uses c in the manners described above. A component graph with these use relations shows how the uses of the original component spread to advices as a result of aspect refactoring.

On the other hand, a directed edge from class c to aspect a represents weaving relationships between c and a, meaning that a has at least one advice that weaves into c. These relations are determined by examining the pointcut specifications in a. Syntactically, the direction of the relationship is from aspect to class, but logically, we can say that class c uses advices from aspect a, hence the direction of the directed edge is from class to aspect. This type of use relation shows how an original class is decomposed into base class and aspects, and how aspects are composed into the base class by refactoring.

2.3 Clone Relation Component Graph

A code clone is a code fragment that has a similar part to it in source code. It is pointed out that code clones make software maintenance difficult[8, 15, 20]. The code clone problem can become serious, especially for large scale software. As with use relations, we model software systems based on clone relations by using a graph. As these are equivalence relationships, we use an undirected graph. A node in the graph represents a software component, and an edge between x and y represents a *clone relation* meaning that both component x and component y have similar code fragments.

Clone relation metrics are calculated using CCFinder[20]. In the graph, we treat Java files as components. Two files have a clone relation if they have similar code fragments that are longer than 25 tokens. In analyses involving aspectized versions, we also take into account clone relations between aspects and between aspects and classes(Java files) to understand how existing code clones in original classes spread to aspects or how clone relations are deleted or created newly by aspect refactoring. In such case, an edge between aspect x and class (or aspect) y represents a clone relation meaning that advices(statements) in aspect x and class y(or advices in aspect y) have similar code fragments that are longer than 25 tokens.

3. EMPIRICAL STUDY

3.1 Purpose of Study

We present case studies of two aspect refactoring projects. In each of these projects, a large Java software system was refactored into AspectJ aspects. By comparing the original Java software structure and the refactored structure, we analyze how the use and clone relation component graphs changed during the refactoring process.

3.2 Preparation

The projects selected for this study are AJHotDraw[1], which is an aspectized version of JHotDraw[2], and a refactoring project[3] for Berkeley DB Java Edition[4]. Each project performed refactoring by extracting cross cutting features in the original system. In both projects, aspect refactoring was carried out in a systematic and disciplined manner, with judicious use of available idioms and patterns. For each project, the changes in use and clone relations are analyzed. These metrics are calculated using SPARS-J and CCFinder and the result is manually organized. We will provide a brief overview of these projects and compare the original and refactored structures.

3.3 AJHotDraw (AJHD)

JHotDraw is a Java-based GUI framework for technical and structured graphics. The AJHotDraw project[21] was formed to identify and evaluate template-based solutions for refactoring object-oriented into aspect-oriented code [16]. It branched off from JHotDraw 6.0 and released three versions, 0.2, 0.3 and 0.4. Crosscutting concerns were extracted in incremental steps, and new aspects were created in each version. The aspectization process was guided by several patterns, as explained in detail in [16]. In the following sections, we briefly outline the aspects introduced in each version and then analyze them in terms of the change in use and clone relations. The JHotDraw aspect refactoring work was small enough that it was possible to use graph visualization to illustrate how affected use and clone relations moved from classes to aspects. As the number of affected relations increased, the visualizations are supplemented by tabulating the classes that were most impacted.

3.3.1 From Ver0.1 to Ver0.2

In version 0.2, 5 aspects were created. Each aspect weaves persistent read and write methods to figure-related classes.

[1]http://swerl.tudelft.nl/bin/view/AMR/AJHotDraw
[2]http://www.jhotdraw.org/
[3]http://wwwiti.cs.uni-magdeburg.de/iti_db/research/berkeley
[4]http://www.oracle.com/database/berkeley-db

Thus, these methods have disappeared from the base because the declaration of Storable interface was moved to the aspects. As a result of this, use relations from such figure-related classes to the classes handling persistent output and input were affected by the refactoring. Figure 1 shows the use relations that were extracted from the base.

Clone relations are also changed. Figure 2 shows the affected part of the clone relation graph. From this we confirm that some clone relations disappeared from modified Figure-related classes. But when we take aspects into consideration, we find that many clone relations have in fact only moved. As shown in Figure 3, we identified clone relations between aspects and existing classes. Specifically, 32 out of the original 37 persistent read and write methods still remain in base classes. Thus, only a limited part of similar code fragments moved to the created aspects. This indicates that the extracted aspects are part of larger features still implemented in the base classes, so these Storable classes continue to be used by many other existing classes.

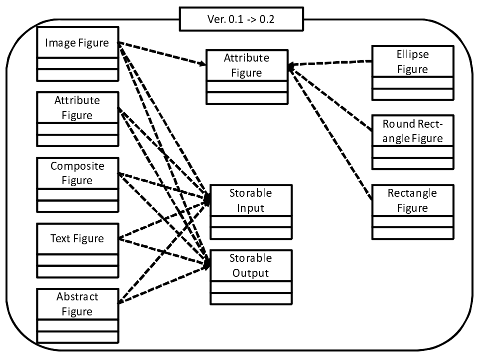


Figure 1: Use relations extracted in Ver. 0.2

3.3.2 From Ver0.2 to Ver0.3

In version 0.3, one empty class named GenericRole and 4 aspects were created. Aside from an aspect (CmdCheckViewRef) that enforces a contract for all AbstractCommand descendants, most aspects added here are concerned with superimposing an observer pattern for notifying and handling changes in selecting figures. GenericRole was extended to encapsulate the observer and subject role. We compare the refactored structure in version 0.3 with the one from version 0.2. Incoming edges to the FigureSelectionListener class were extracted as in Figure 4.

Changes to the clone relations are shown in Figure 5. At first, 7 classes are strongly connected, however, we find that this clone group is now decomposed into several smaller clone groups after a common method-call statement was extracted. When we take into account advices in the aspects, we identify one new clone relation, between StandardDrawing and FigureSelectionSubjectRole.aj, resulting from the extraction of a method in StandardDrawing that has similarities with another method in this class. Other new aspects have no clone relations with classes in the clone group. This is perhaps because all of classes in the group are modified at once and the methodology of the refactoring is not based on method extraction, but statement extraction.

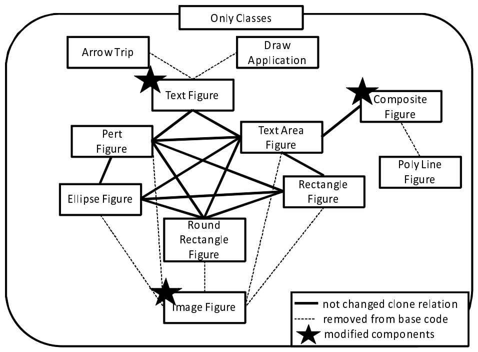


Figure 2: Clones changed in Ver. 0.2(only classes)

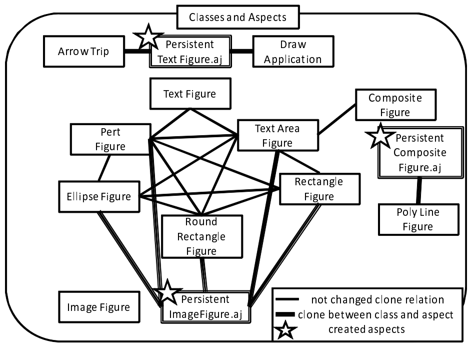


Figure 3: Clones in Ver. 0.2(classes and aspects)

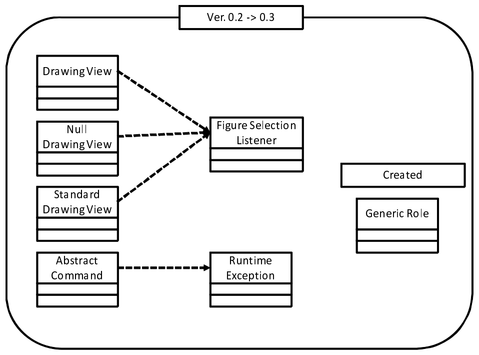


Figure 4: Use relations extracted in Ver. 0.3

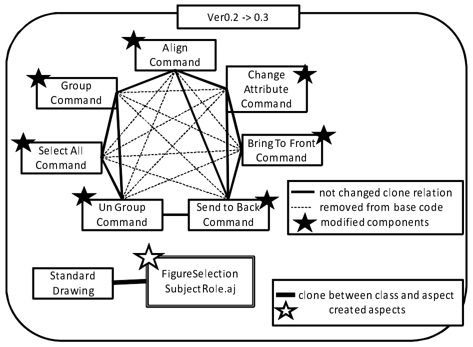


Figure 5: Clones changed in Ver. 0.3

3.3.3 From Ver0.3 to Ver0.4

In version 0.4, one aspect was deleted and 22 aspects were created. In this update, several features, such as handling IO exception, CommandListener, undo handling, were extracted as aspects and existing aspects were also reorganized. The biggest extraction is undo handling; undo-related classes were aspectized, with undo features in each class extracted as aspects. Regarding the aspectization approach, the pattern of extending GenericRole was also used as in the case of version 0.3. Moreover, we can find other weaving patterns in class related with CommandUndo and ToolUndo. These details are different, but all of these aspects weave similar methods and advices to corresponding classes.

In this version, 10 classes and 96 use relation edges are removed and 5 edges are added compared to version 0.3. Deleted classes are mainly UndoActivity classes that had been inner classes for each command class. To check the use relations that were removed, the reduction in incoming and outgoing edges of each component on the component graph is analyzed. Table 1 is a list of components whose outgoing edges decreased. Decrease in outgoing edges of a component implies that the component has stopped using some classes. From Table 1, not only deleted inner classes, Command-related and Tool-related classes are on the list. This was because, in addition to undo features, other undo-related methods in each command class were also extracted and moved to aspects.

Table 2 is a list of components whose incoming edges decreased. Decrease in incoming edges of a component implies that some classes have stopped using this component. From Table 2, undo-related classes, such as UndoableAdapter, Undoable are on the list, and commonly used classes such as DrawingView, Figure, FigureEnumeration and so on are also on the list.

Table 1: The change of outgoing edges (0.3 and 0.4)

Class	0.3	0.4	change
GroupCommand$UndoActivity	9	removed	-9
CutCommand$UndoActivity	8	removed	-8
UndoableCommand	8	removed	-8
standard.CutCommand	11	6	-5
standard.DeleteCommand	10	5	-5
AlignCommand$UndoActivity	5	removed	-5
DeleteCommand$UndoActivity	5	removed	-5
PasteCommand$UndoActivity	5	removed	-5
figures.ConnectedTextTool	12	8	-4
figures.TextTool	15	11	-4

The affected clone relations are shown in Figure 6. A large number of clone relations disappeared, with all of clone relations removed from some modified classes. However, when we take into account advices in the aspects (Figure 7), we find that most clone relations simply moved to the related aspects as in the case of ver 0.2, and only 6 clone pairs actually disappeared. This implies that extracted undo-related code have similarities to other fragments that remained in the base classes. This is confirmed by an inspection of the code from which we find 19 out of the original 26 UndoActivity classes are still in the base code.

Altogether, the number of clone relations decreased from

Table 2: The change of incoming edges (0.3 and 0.4)

Class	0.3	0.4	change
util.Undoable	48	35	-13
framework.DrawingView	131	121	-10
util.UndoableAdapter	33	24	-9
PasteCommand$UndoActivity	7	removed	-7
framework.Figure	135	129	-6
framework.FigureEnumeration	73	68	-5
util.CommandListener	4	0	-4
standard.FigureTransferCommand	10	6	-4
standard.FigureEnumerator	24	21	-3
util.CollectionsFactory	47	44	-3

206 in JHotDraw to 158 in AJHotDraw, so 23% of clone relations are affected by these refactorings, and 1 new clone relation between class and aspect is created. Of the 48 affected clone relations, 16 are actually removed, 29 clone relations are now between aspects and classes, and 3 clone relations are purely between aspects.

3.4 Aspect Refactoring of Berkeley DB (ABDB)

Berkeley DB Java Edition (BDB JE) is a software component library that provides a high-performance embedded database. The BDB JE refactoring project [9] set out to investigate the use of AspectJ to implement a product line consisting of a common base and several optional features that can be composed together. The refactoring was carried out by identifying several features from BDB JE and then refactoring each into a combination of classes and associated aspects.

We downloaded the publicly available refactored code which consists of a base program and 28 aspectized features, containing a total of 107 aspects. And then, we extracted the set of Java classes from it, and compared the class structure of the refactored Java code with the original BDB JE code.

For the original BDB JE, we selected, among the versions available on the BDB website, version 2.1.30, whose class structure and source code descriptions are most similar to the refactored code. To give the original version a resemblance to the refactored version, we removed some packages, such as test, example, collection, JEC and so on, from the original version. We also found that some packages were renamed during refactoring project. However, if these renamed packages have approximate counterparts in the original version, we treat these packages as the same. In such cases, we call such components by the name in version 2.1.30.

In what follows, we call the refactored system, ABDB, and call the original (version 2.1.30), JBDB. We analyze the change in use and clone relations between JBDB and ABDB. We analyze the data by tabulating the classes that were most impacted with respect to changes in incoming and outgoing use relations as well as clone relations. We also check if the differences in use relations between JBDB and ABDB are statistically significant. Finally, we manually inspect the impacted classes to find characteristics common to these classes.

3.4.1 *Impact on Use Relations*

Reduction in outgoing edges

As in the case of AJHD, the reduction in outgoing edges is a good indicator to assess how much of the use of other components was moved to aspects. The reduction in outgoing edges of each component is analyzed by comparing the base class structures of ABDB and JBDB. In terms of the component graph for use relations, JBDB has 331 nodes and 1977 edges while the base class structure of ABDB has 336 nodes and 1681 edges. Thus, about 15% of edges are affected by the refactoring. Furthermore, a one-tailed paired Wilcoxon-Mann-Whitney test[5] of the distributions for outgoing edges of ABDB and JBDB classes also reports that the distribution of outgoing edges in JBDB classes is greater than the ones corresponding to classes in ABDB ($p < 0.0001$).

Figure 8 is a histogram of the outgoing edges of each class for JBDB and ABDB, (the third item is also ABDB, but includes outgoing edges to aspects, and is explained later). From the graph, we can confirm that a number of components that have no outgoing edges or have a low degree of outgoing edges increased after the refactoring. On the other hand, a number of components that have a high degree of outgoing edges decreased slightly overall. We consider these affected components are not only classes which

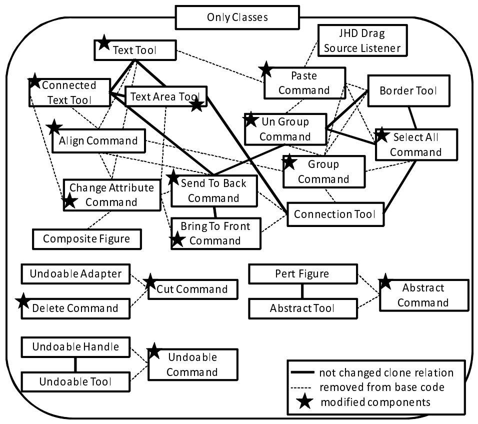


Figure 6: Clones changed in Ver. 0.4(only classes)

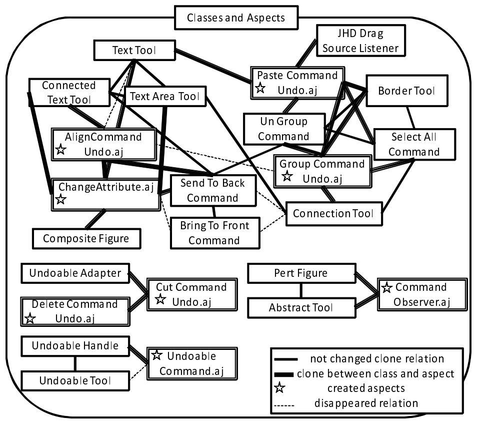


Figure 7: Clones in Ver. 0.4(classes and aspects)

[5]The non-parametric Wilcoxon-Mann-Whitney test [19] was used here and in subsequent statistical testing because the distributions being compared were not normally distributed, as confirmed by the Shapiro-Wilk test [18] for normality.

implement the extracted feature, but also local or core components which use the extracted feature.

Table 3 is a list of components whose outgoing edges decreased significantly. We observe that some DB-related classes which use a lot of another classes are deleted during refactoring. Impl and Manager classes organize and control some major features, and Environment, Tree, and Database classes are part of the BDB API. In JBDB, such management and API classes used various features, which were extracted in the refactoring.

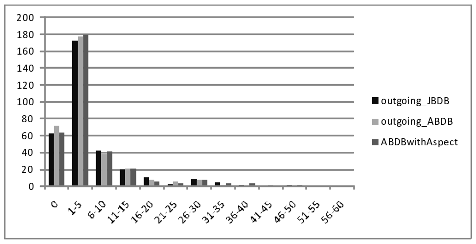


Figure 8: Distribution of number of outgoing edges

Table 3: The change of outgoing edges

Class	JBDB	ABDB	change
dbi.DatabaseImpl	45	23	-22
utilint.DbScavenger	22	removed	-22
dbi.EnvironmentImpl	54	34	-20
util.DbRunAction	18	removed	-18
txn.TxnManager	18	8	-10
util.DbLoad	10	removed	-10
util.DbCachesize	10	removed	-10
log.FileManager	28	19	-9
tree.Tree	50	41	-9
Database	36	28	-8
Environment	28	20	-8

Reduction in incoming edges

As in the case of outgoing edges, reduction in incoming edges of each component is analyzed by comparing the Java class structures in ABDB and JBDB. The reduction in incoming edges is a good indicator to assess how much of the usage of a component was moved to aspects. We also performed a Wilcoxon-Mann-Whitney test on the distributions for incoming edges of ABDB and JBDB as in the case of outgoing edges. It also reports that the distribution of JBDB's incoming edges is significantly greater than the one of ABDB's incoming edges ($p < 0.0001$). This shows that the scale of the refactoring was large enough to affect the distribution of incoming edges.

Figure 9 is a histogram of incoming edges of each class for JBDB and ABDB (the third item is also ABDB, but includes incoming edges from aspects, and is also explained later). From the graph, we can confirm that a number of components that have no outgoing edges increased through the refactoring. On the other hand, a number of components that have a high degree of outgoing edges also decreased slightly overall. We consider classes whose incoming edges are completely removed are mainly classes for implementing extracted features, and classes whose incoming edges are slightly decreased are mainly classes used by the extracted features.

Table 4: The change of incoming edges

Class	JBDB	ABDB	change
dbi.MemoryBudget	31	0	-31
utilint.Tracer	23	removed	-23
latch.LatchSupport	24	6	-18
StatsConfig	19	1	-18
Transaction	14	0	-14
EnvironmentStats	12	0	-12
latch.Latch	15	4	-11
LockStats	11	1	-10
Environment	19	10	-9
DatabaseException	71	63	-8

Table 4 is a list of such components. Classes central to extracted features, such as MemoryBudget, Latch, and Transaction, are on the list. In addition, classes to collect and hold the execution information are also on the list. These components' features were extracted during the refactoring project, and codes and methods which use these features were also extracted.

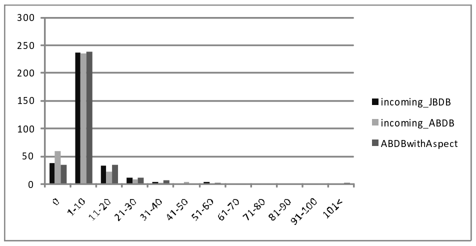


Figure 9: Distribution of number of incoming edges

Including use relations from/to aspects

Next, we checked the impact on the component graph when aspects are included as components. In this analysis, we considered two types of use relations: from class to aspect and from aspect to class.

First, we consider use relations from class to aspect, as defined in Section 2.2, that is, weaving relations between a class and the aspects that weave to it (as specified in the pointcut). In object-oriented systems, to understand how a class works, we often check how it uses other classes. Similarly for an aspect-oriented system, we will check not only classes used by a class, but the aspects it uses as well. We treat this weaving relation as a use relation from class to aspect, and we checked number of outgoing edges from each class. This gives a rough estimation of the effort for understanding how the class works.

In total, we find 260 such use relations to 107 aspects in ABDB. Table 5 is the list of classes most frequently targeted by aspects. Implementation classes, such as EnvironmentImpl and CursorImpl, management classes, such as FileManager are on the list. Moreover, classes handling information on the execution, such as Params and Stats, and main classes for certain database helper functions, such as Cleaner and Checkpointer are also on the list. This implies that these classes use the features extracted by these aspects, or the extracted features use these classes to store and handle related information.

The third item in Figure 8 is a distribution of outgoing edges of each class for ABDB with aspects. The number of classes who have no outgoing edges are almost same as the original JBDB, and the Wilcoxon-Mann-Whitney test indicates that the difference between the two distributions is not statistically significant ($p = 0.6086$). Thus we can confirm that uses of other class are completely replaced by the aspect weaving in some classes.

Table 5: How many aspects weave to each class?

Class	Aspects
dbi.EnvironmentImpl	24
Environment	15
log.FileManager	11
dbi.CursorImpl	10
config.EnvironmentParams	9
Database	8
EnvironmentStats	8
recovery.Checkpointer	7
cleaner.Cleaner	7
dbi.DatabaseImpl	7

Next, we consider use relations from aspects to classes, counting how many aspects use each target class in their advices. Determining how a class is used or referred to by other classes is also an important activity to understand how it works. In the case of aspect-oriented systems, such information spreads also into advices in several aspects. So we treat such use relation in advices in each aspect as use relation from the aspect to class, and we checked the total number of incoming edges to each class. It gives a rough estimation of the effort for understanding how the class is managed in the system.

Table 6: How much is each component used by aspects?

Class	Aspects
DatabaseException	81
dbi.EnvironmentImpl	60
StatsConfig	23
dbi.DatabaseImpl	20
tree.IN	20
dbi.DbConfigManager	17
latch.LatchSupport	17
dbi.MemoryBudget	17
EnvironmentStats	16
config.EnvironmentParams	15

In this way, we identify 700 edges from 107 aspects to classes. The third item in Figure 9 is a distribution of incoming edges of each class for ABDB with aspects. The number of classes who have no incoming edges are almost same as the original JBDB's one, and in fact, the Wilcoxon-Mann-Whitney test indicates that the number of incoming use relations has increased and this increase is statistically significant ($p < 0.0001$). Thus we can confirm that most of the extracted use relations are moved to aspects and new use relations are fanned in from multiple aspects.

Table 6 is a list of classes with the most incoming edges from aspects. In general, methods implementing a feature are more likely to call other methods that are also part of that feature, thus if the feature is refactored into aspects, one side of the use relations are moved to aspects. We find a large number of aspects calling feature code still in base feature classes such as LatchSupport and MemoryBudget or in classes that handle execution information such as StatsConfig and EnvironmentStats.

Table 7: The change of incoming edges from classes and aspects

Class	JBDB	ABDB			change
		Class	Aspect	Total	
Tracer	23		removed		-23
dbi.MemoryBudget	31	0	17	17	-14
TreeWalkerStats-Accumulator	8		removed		-8
Transaction	14	0	7	7	-7
LockStats	11	1	4	5	-6
EnvironmentConfig	16	10	1	11	-5
config.ConfigParam	16	12	0	12	-4
DbInternal	19	12	3	15	-4
DatabaseStats	6	2	2	4	-2
latch.SharedLatch	6	2	2	4	-2

Class	JBDB	ABDB			change
		Class	Aspect	Total	
DatabaseException	71	63	81	144	73
dbi.EnvironmentImpl	95	93	60	153	58
tree.IN	45	44	20	64	19
OperationContext	new	12	6	18	18
dbi.DatabaseImpl	55	53	20	73	18
tree.BIN	20	20	12	32	12
dbi.DbConfigManager	27	22	17	39	12
txn.Locker	39	38	12	50	11
dbi.CursorImpl	16	14	13	27	11
EnvironmentParams	28	24	15	39	11

However, it is striking that 81 of 107 aspects use DatabaseException in their advices for exception handling, and 60 aspects use EnvironmentImpl in their advices. EnvironmentImpl encapsulates an entire execution, so the class is used frequently as an argument passed by methods, and some of its members are accessed in the method.

Table 7 lists the classes that have a large difference in incoming edges when edges from aspects are accounted for. The upper table is in ascending order. Classes for a certain extracted feature, such as Transaction and MemoryBudget and so on, and classes encapsulating execution information, such as StatsConfig, EnvironmentConfig and so on, are on the list. The use relations of such classes were reduced, so we can confirm that refactoring into aspects managed to successfully encapsulate the extracted features and data structure.

The lower table is in decreasing order of difference. Implementation classes such as EnvironmentImpl, DatabaseImpl, CursorImpl and classes for the B-tree such as IN, Tree are on the list. These components increase in number of incoming edges after refactoring and are used frequently in argument passing of methods. Such classes that control some major features are used widely by both remaining classes and extracted aspects, so users of the class are scattered over the software system.

Analyzing distribution of edges related to aspects

Next, we compare the change between edges in ABDB's base structure (without aspects) and the one in JBDB from the change between edges in ABDB's structure with aspects and the one in JBDB.

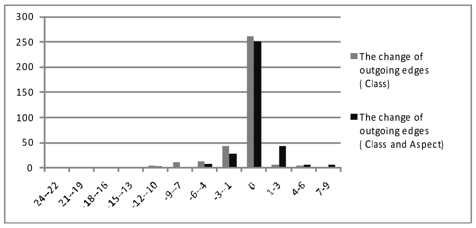


Figure 10: Distribution of the change of outgoing edges

Figure 10 shows a distribution of the change of outgoing edges for each class. From this graph, we can confirm the following two things; By comparing only in the class structure (the first item), we can confirm some outgoing edges are extracted. By comparing the total structure(with aspects), that is the second item, we can confirm the total number of outgoing edges are not changed so much. From a statistical testing about distributions for outgoing edges of JBDB and ABDB with aspect, we cannot confirm that the distribution of JBDB's outgoing edges is greater than the one of outgoing edges of ABDB with aspects. The use of other classes are moderately replaced to the weaving by aspects, and total effort to check based on outgoing use relations would be not changed so much.

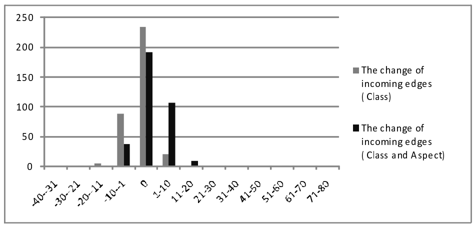


Figure 11: Distribution of the change of incoming edges

Figure 11 shows a distribution of the change of incoming edges for each class. From this graph, we can confirm the following two things; By comparing only in the class structure (the first item), we can confirm some incoming edges are extracted from a lot of classes. However, by comparing the total structure(with aspects) from the original structure, we can confirm the total number of incoming edges tend to increase in the distribution. From a statistical testing, we can confirm that the distribution of incoming edges of ABDB with aspects is greater than the one of JBDB's incoming edges. So, there is a difference between distribution of the change of outgoing edges and the change of incoming edges; total effort to check based on used-by relations may increase.

Figure 12 is a distribution of changes of outgoing and incoming edges for each class. The x-axis plots the change between edges in ABDB's base structure and the one in JBDB, and the y-axis plots the change between edges in ABDB's structure with aspects and the one in JBDB.

We observe that many components are near the graph origin; these components were not affected by the refactoring. Figure 12 also shows that a large majority of components are in the left side of the y-axis, indicating that refactoring simplified use relations in the base structure. Components in the right side of the y-axis are mostly newly created. Some components are plotted along the line of $x = y$, this is because there are created classes through refactoring.

We observe that many components are below the x-axis, that is, their total edges decreased even when use relations from aspects are taken into consideration. These include removed components, but also for components of certain extracted features. However, we observe that a lot of components are above the x-axis, and especially for incoming edges, we can confirm many components are plotted in the upper side of the graph. It indicating that their total edges are almost the same or increase when use relations from aspects are accounted for, however, incoming edges to classes are easy to increase. Total edges are not necessarily reduced even if developer removes a lot of use relations from the target class. This indicates that, for many components, they got codes from aspects in response to the number of extracted features that they use, or their usage was dispersed into several similar aspects.

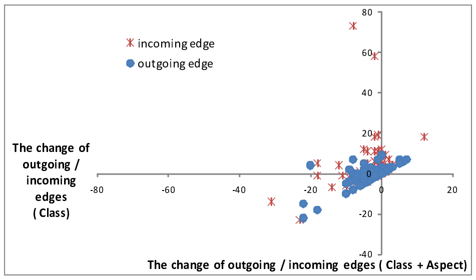


Figure 12: The distribution of change of edges

3.4.2 *Impact on Clone Relations*

We also compared the clone relation graphs of JBDB and ABDB. When considering only classes, there are 277 nodes and 365 clone relations in the component graph of JBDB, and 282 nodes and 158 clone relations in ABDB. During this refactoring, 13 Java files are removed and 18 Java files are added. Some clone relations are also removed and others created; 218 clone relations are removed and only 11 clone relations are created, while the remaining 147 clone relations are not affected. In this way, a large percentage of clone relations disappear from the class structure. Table 8 lists classes which lost the highest number of clone relations. Central classes of certain features, such as Checkpointer, Cleaner, DbLoad, Tracer, Evictor, and Txn, and several Manager classes are on the list. This indicates that a lot of similar code fragments are extracted from these files.

Next, we consider clone relations when aspects are accounted for. In our analysis, we identified 81 new clone relations, 35 between class and aspect, and 46 between aspects. Thus 218 clone relations removed from the class structure are replaced by a much smaller set of 81 clone relations. This indicates that over half of clone relations removed from class structure are actually removed. We can still find some strongly connected clone groups consisting of classes and aspects in the graph. But overall, this refactoring was very effective from the perspective of clone removal.

Table 8: The change of clone relations

File	JBDB	ABDB	change
Checkpointer	34	7	-27
Cleaner	28	1	-27
RecoveryManager	31	8	-23
TxnManager	20	0	-20
DbLoad	19	removed	-19
Tracer	17	removed	-17
EnvironmentStats	17	0	-17
Evictor	20	16	-16
IN	26	10	-16
Txn	19	4	-15

4. DISCUSSION

We revisit our research questions from Section 1, discussing the effectiveness and characteristics of aspect-oriented refactoring. We answer the specific questions first and then summarize the discussion to answer the main questions.

4.1 Effectiveness of Refactoring

4.1.1 Q1-1: Is refactoring effective for reducing use and clone relations between classes?

From Figure 12, we observe that the use relations between classes (x-axis) are mostly reduced after refactoring. This reduction is also confirmed to be statistically significant. At the component level, use-relations which form coupling relations were moved out of the base code as indicated by Figures 1 and 4, so refactoring contributes to the simplification of use relations between classes. We consider that refactoring of features into aspects is effective if the feature already has a localized class structure. In this case, not only the feature itself, but also statements using the feature are moved to aspects cleanly. In this way, such relations between classes are simplified, and refactoring contributes to modularizing scattered information.

From the result, we also find that scattered, similar code are extracted to aspects, so clone relations are also reduced after refactoring, especially classes which become a core of certain features.

4.1.2 Q1-2: Is refactoring effective for reducing total use and clone relations (classes and aspects)?

As shown in Figure 12 and Tables 6 and 7, the refactoring may cause an increase in incoming use relations if components are widely used in aspects and classes. In ABDB, such components are frequently used as arguments. In general, classes like these are a global resource of the system; the class is used widely by both remaining classes and extracted aspects, so use relations to the class are scattered over many components in the system. This phenomenon may make it more difficult to understand the global behavior of the system when aspects are considered. The additional couplings also make it more difficult to evolve the behavior of core components. The BDB refactoring project [9] also reported several limitations in refactoring using AspectJ, for example, it was impossible to add new exceptions to the existing methods, so they declared all throwable exceptions of all features in the base code. As a result, numerous aspects had to reference base classes such as DatabaseException, thus driving up the number of incoming use relations to such classes (see Table 7).

Figure 13 shows the distribution of changes of incoming edges from classes, and from classes and aspects for three versions of AJHD. In the graphs, almost all of the components are around the origin and x-axis, so use relations did not increase significantly.

As discussed in the AJHD project [16], modifications to the base requires an awareness of the advice that applies to it. The ABDB project [9] also reported similar issues. In spite of their best efforts, they were dissatisfied with the results because maintainability, fragility and code readability became worse. We think the propagation of use relations in aspects contributes to the dissatisfaction.

From the perspective of clone reduction, ABDB's refactoring effort was quite successful. Over 100 clone relations were actually removed, with a small number that moved to aspects. Some clone relations between aspects were also produced when similar code fragments were extracted into different aspects; in many cases, this situation cannot be avoided if feature boundaries need to be preserved.

As shown in Figure 14, a clone relation between class and aspect forms when only one part of a clone group is extracted. This means that the scope of the classes considered for aspect extraction could have been extended so as to include all clone fragments. If a size of the scope was adequate, the number of clone relations would decrease or the number of clones between aspects would increase as in the case of ABDB.The opposite can be said for AJHD. Most clone groups were only partially moved to aspects, implying that perhaps other design considerations had to be taken into account in determining the scope of extraction.

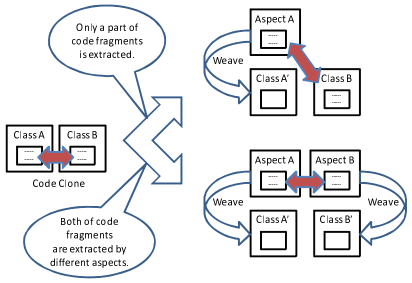
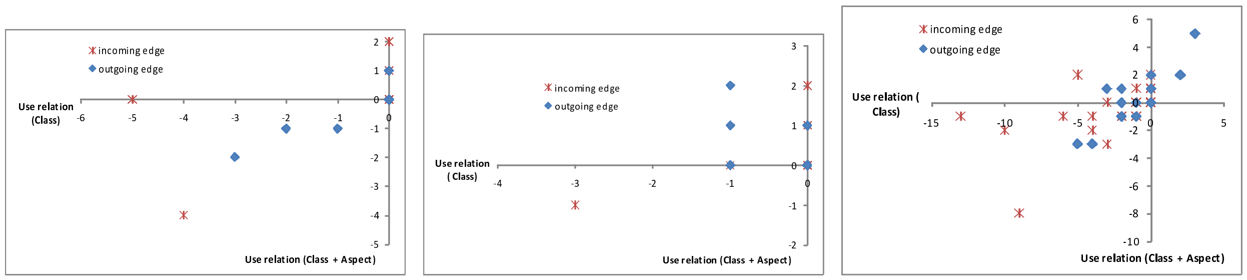

Only a part of code fragments is extracted.
Aspect A
Weave
Class A'
Class B
Class A
Class B
Code Clone
Aspect A
Aspect B
Weave
Weave
Class A'
Class B'
Both of code fragments are extracted by different aspects.

Figure 14: Extraction of code fragments in different classes

If developers need to modify extracted code fragments after refactoring, it may be better to modify clones from two aspects than having to modify clones in both class and aspect. If there is a clone relation between class and aspect, a developer may need two kinds of modifications, one to the class and the other to its clone pair in the aspect. If the aspect is not conceptually related to the class, it would be easy to neglect to modify one or the other, particularly over time or with new developers. On the other hand, in the case of clone relations between aspects, we speculate that the aspect-to-aspect symmetry as in the scenario depicted in Figure 14 would lead developers to check similar aspects when making updates, so clones would not be as likely to be neglected.

4.1.3 Q1: Is aspect-oriented refactoring effective for improving modularity and complexity?

In general, refactoring by aspects improves the base code's



(a) AJHD Version 0.2 (b) AJHD Version 0.3 (c) AJHD Version 0.4

Figure 13: The distribution of change of incoming edges from classes and aspects for AJHotDraw

modularity by reducing the number of use relations. However, care must be taken to pay attention to the dispersing of use relations to aspects. This hinders the maintainability of the base as changes to classes can have an impact on the aspects which use it. If inadequate attention is paid to use relations from aspects, the maintenance of some components can become more complex and error prone. To mitigate the increase of use relations, an object-oriented refactoring of the code may split up some of the highly used classes. An object-oriented refactoring prior to aspect-oriented refactoring was also suggested in [16]. In many cases, the increase in use relations is unavoidable when the refactoring project is constrained to preserve existing component interfaces and work with some of the existing object designs.

Refactoring by aspects also simplifies the base code's complexity by factoring out clones. But from the results of two projects, aspect refactoring may move clone relations to aspects. To sustain maintainability, developers should determine the scope of extraction carefully. If the scale is too small, this will be reflected directly in the number of code clones between class and aspect. A finer-grained approach, based on extracting statements rather than methods, as performed in AJHD ver 0.3, is also effective to minimize propagation of clones. However, the granularity may be too small to apply on a large scale. A complementary approach is to use a clone detection tool like CCFinder [23] to guide developers in determining the proper scope by identifying all clone relations in the code.

While we can get the desired result if the refactoring is done perfectly, it seems that there is a very high bar to complete it in real software systems. We need experts in aspect refactoring as well as experts who have a global understanding of the system. Without enough preparation, the refactoring produces suboptimal results. However, if we can characterize classes that are suitable for aspect refactoring it may be possible to develop analysis tools that can identify the best candidates for refactoring. We discuss such characterizations next.

4.2 Classes Suitable for Aspect Refactoring

4.2.1 *Q2-1: What kinds of components are likely to have fewer use relations after refactoring?*

Figures 1 and 4, and Tables 1 and 3 indicate that outgoing edges for using the extracted features are removed from class structure. If the extraction is conducted on a large scale, use relations of management classes, such as Impl and Manager classes, and API classes, are greatly simplified.

A lot of aspects weave advices to management classes and API classes as shown in Table 5. In the advices, statements call a lot of classes central to extracted features, as well as stats classes holding execution information, as shown in Table 6. Management classes and API classes are also used in the advices and methods, often passed in as arguments. From Table 7, such components are still used in the remaining classes.

Figures 1 and 4, and Tables 2 and 4 indicate that incoming edges to components whose features are extracted are removed from class structure. If the extraction is conducted on a large scale, use relations to management classes and API classes are also removed, however, the reduction in number of incoming edges for such classes is not significant.

4.2.2 *Q2-2: What kinds of components are likely to have fewer clone relations after refactoring?*

Results of AJHD (Figure 2,5, and 6) indicate a significant reduction of clone relations between classes that implement similar features. Clone relations completely disappear when similar crosscutting code are extracted from all applicable classes at the same time. Otherwise, clone relations remain and get moved to relations between classes and aspects, like undo related aspects in ver 0.4 (Figure 7).

Results from ABDB (Figure 8) show a large reduction in clone relations from classes that implement similar features, indicating a significant amount of similar code crosscutting these classes.

4.2.3 *Q2: What are the characteristics of classes likely to be strongly affected by such refactoring activities?*

In general, the number of use relations between classes decreases as a result of aspect refactoring. The reduction of edges arises on the class around the features extracted by aspect refactoring. If the extraction is conducted on a large scale, not only the components which implements or uses certain features, but also the management classes and API classes will see a reduction in use relations.

In this study, we also confirm a reduction of clone relations mainly on the classes which implement similar features. Similar code fragments form readily in such classes because similar features tend to have similar code, requiring similar behavior as well as pre- and post-processing. Through aspect refactoring, code for such processing can be factored out into aspects.

4.3 Threats to Validity

4.3.1 Construct validity

Construct validity refers to how well we have measured the qualities under study, namely modularity and complexity. Modularity was examined from the perspective of coupling, specifically usage dependencies; the fewer dependencies we have, the more modular the system. It can be argued that the use relations identified may not adequately capture all usage dependencies, such as inherited dependencies due to polymorphism as well as type parameter dependencies due to generics. Inherited dependencies are an example of indirect relationships. We have conducted additional analyses using component ranking [7] which can account for such indirect relationships. The results when indirect relationships were taken into account are consistent with our findings here. Generics were not used in the versions we examined of the two projects.

Complexity was examined from the perspective of difficulty of maintenance, specifically due to clones. It can be argued that the clones identified may not capture all clone dependencies, especially clones with less than 25 tokens in common. We tried several thresholds before settling on 25 tokens since, from the data, this was the typical size of an exception block, which was a commonly aspectized code fragment.

4.3.2 Internal validity

Internal validity refers to the ability to make causal conclusions between treatments and effects, in this case, between aspect-oriented refactoring and modularity and code consolidation. By comparing the aspect-oriented and non-aspect-oriented version of the same project, we controlled for differences in the application complexity. Thus, these versions act as natural controls for making controlled observations [14]. Also, in all the analyses, we made certain to explain the results with corroborating observations of examples from the projects. Furthermore, when possible, we highlighted the consistency of our findings with the results the refactoring teams also arrived at.

4.3.3 External validity

External validity refers to the ability to generalize the results of our study, in this case, whether the projects chosen are representative of industrial systems. Unlike toy examples, both JHotDraw and BDB are relatively large projects with a history of accommodating real-world needs. This is attested to by the fact that they are used by a number of other industrial applications. Obviously, they do not reflect the wide diversity of all real-world projects. Therefore, further replications are needed to verify if our conclusions hold with additional data from other aspect-refactoring projects. The consistency of results between our 2 large projects gives us some confidence that other refactored systems will also have the similar results.

4.4 Related Works

Our work is the first comparison study of large scale aspect-oriented refactoring projects. Other researchers have studied impacts of object-oriented refactoring in real-world systems. For example, Du Bois, et al. [2] investigated the impact of standard refactoring activities (e.g., Move Method, Extract Class, etc.) on coupling and cohesion in Apache Tomcat. Their results conclude that refactoring opportunities that improve coupling and cohesion are hard to find; for many forms of refactoring, couplings just move elsewhere. Our findings also indicate that use relations mostly do not disappear; they just move into aspects.

Our work also complements a growing body of research assessing the benefits that aspect-oriented systems offer to subsequent maintenance and evolution activities [1, 3, 22]. It is our hope that our work on use and clone relation change can provide additional quantitative information for explaining the results of such studies.

Eaddy, et al. [5] proposed a set of concern metrics for precisely measuring scattering and tangling. While their work is mostly predictive in nature and useful for detecting crosscutting code, our work is retrospective, studying crosscutting code that is already in aspects. This suggests a future thread of investigation studying how much of the crosscutting code suggested by such concern metrics were actually refactored into aspects.

Our empirical study approach is similar to Kulesza, et al. [12], which investigated the effectiveness of designing based on aspect oriented languages. In their experiments, they compared aspect-oriented and object-oriented implementations of the same system using several general metrics. They also analyzed a refactored structure without the aspects, as we did here. While their results were mostly in favor of the aspect-oriented implementation, we note that the system was smaller, which was adequate for the purpose of their experiments comparing the performance of maintenance tasks on both implementations. We also note that the aspect-oriented implementation was designed from the ground up, following the same principles as the object-oriented design but not constrained to preserve specific component interfaces or subsystem boundaries of the object-oriented implementation.

Our work on clone relation analysis is also similar to Yoshida, et al. [23], where CCFinder was used to detect sets of clones in order to inform the refactoring process. Case studies were conducted to detect such clones in large systems such as ANTLR, Apache Tomcat and JBoss. We extend their results by examining how detected clone groups are actually reflected in aspects after refactoring.

5. CONCLUSION

In this paper, we examined the effectiveness of aspect-oriented refactoring in improving code modularity and complexity through factoring out crosscutting code from existing non-aspect-oriented systems. We conducted a multiple-case study on two large projects. For each project, we compared an original Java software structure and the refactored structure with respect to changes in use and clone relations. Even though these projects had different purposes, they reached similar conclusions regarding the benefits and limitations of aspect-oriented refactoring. Our analysis adds to their contributions by providing quantitative confirmation of their evaluation results.

We confirmed that refactoring causes propagation of use relations for a certain set of classes. In general, such classes are global resources of the software system, used widely by remaining classes and extracted aspects. Hence, use relations to such classes are scattered all over the software system. Thus, while the overall modularity of the base Java code appears to improve, coupling increases when accounting for use relations from aspects. On the other hand, we also confirmed the number of clone relations between class and aspect increases if only one part of a clone group is extracted. These hinder the maintainability of the base as changes to classes can have a negative impact on the classes and aspects which use it. Some of the problems may be attributed to the limitations of AspectJ, as pointed out by the developers of AJHD and ABDB. We also attribute these difficulties to the scope of the original code considered for each extraction. In many cases though, this is unavoidable, especially when the refactoring project is constrained to preserve existing component interfaces and to work with some of the existing object designs.

As future work, we plan to apply our analysis to other aspect refactoring projects. In addition to component relationships, other types of metrics related to cohesion and concern diffusion similar to those used by Garcia, et al. [6] will be studied as well. We would also like to analyze whether certain types of object-oriented refactoring can prepare a system for aspect-oriented refactoring. Finally, we also plan to investigate the link between aspect refactoring patterns and patterns of change in relations.

6. REFERENCES

[1] M. Bartsch and R. Harrison. "An exploratory study of the effect of aspect-oriented programming on maintainability". *Software Quality Journal*, 16(1):23–44, 2008.

[2] B. D. Bois, S. Demeyer, and J. Verelst. "Refactoring - Improving Coupling and Cohesion of Existing Code". In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 144–151, 2004.

[3] Y. Coady and G. Kiczales. "Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code". In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 50–59, Boston, MA, USA, 2003.

[4] A. Colyer and A. Clement. "Large-scale AOSD for Middleware". In *Proceedings of the 3rd international Conference on Aspect-Oriented Software Development*, Lancaster, UK, 2004.

[5] M. Eaddy, A. Aho, and G. C. Murphy. "Identifying, Assigning, and Quantifying Crosscutting Concerns". In *Proc. of the First International Workshop on Assessment of Contemporary Modularization Techniques*, page 2, Washington, DC, USA, 2007.

[6] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. "Modularizing design patterns with aspects: a quantitative study". In *Proceedings of the 4th international Conference on Aspect-Oriented Software Development*, Chicago, Illinois, USA, 2005.

[7] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto. "Ranking Significance of Software Components Based on Use Relations". *IEEE Transactions on Software Engineering*, 31(3):213–225, 2005.

[8] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones". In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105, 2007.

[9] C. Kastner, S. Apel, and D. Batory. "A Case Study Implementing Features Using AspectJ". In *Proceedings of the 11th International Software Product Line Conference*, pages 223–232, Kyoto, Japan, 2007.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lope, J.-M. Loingtier, and J. Irwin. "Aspect-Oriented Programming". In *Proceedings of ECOOP '97*, pages 220–242, 1997.

[11] C. Krueger. "Software Reuse". *ACM Computing Surveys*, 24(2):131–183, 1992.

[12] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. von Staa, and C. Lucena. "Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study". In *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM06)*, pages 223–233, Philadelphia, PA, USA, 2006.

[13] R. Laddad. "Aspect-oriented Refactoring: Taking Refactoring to a New Level. ". In *Tutorial at the International Conference on Aspect-Oriented Software Development (AOSD2005)*, Chicago, USA, 2005.

[14] A. S. Lee. "A scientific methodology for MIS case studies". *MIS Quarterly*, 13(1):33–50, 1989.

[15] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code". *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.

[16] M. Marin, A. Deursen, L. Moonen, and R. Rijst. "An integrated crosscutting concern migration strategy and its semi-automated application to JHotDraw". *Automated Software Engineering*, 16(2):323–356, 2009.

[17] W. F. Opdyke. *"Refactoring object-oriented frameworks"*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[18] S. Shapiro and M. Wilk. "An analysis of variance test for normality (complete samples)". *Biometrika*, 52(3-4):591–611, 1965.

[19] S. Siegel and J. Castellan. *"Nonparametric Statistics for The Behavioral Sciences"*. McGraw-Hill, 1988.

[20] T.Kamiya, S.Kusumoto, and K.Inoue. "CCFinder: A multilinguistic token-based code clone detection system for large scale source code". *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[21] A. van Deursen, M. Marin, and L. Moonen. "AJHotDraw: A showcase for refactoring to aspects". In *Proceedings of the Workshop on Linking Aspects and Evolution. 4th International Conference on Aspect-Oriented Programming*, Chicago, USA, 2005.

[22] R. Walker, E. L. A. Baniassad, and G. C. Murphy. "An Initial Assessment of Aspect-oriented Programming". In *Proc. of the 21st International Conference on Software Engineering*, pages 120–130, 1999.

[23] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. "On Refactoring Support Based on Code Clone Dependency Relation". In *Proceedings of the 11th IEEE International Software Metrics Symposium*, page 16(10 pages), Como, Italy, 2005.