Assessing the Impact of Framework Changes Using Component Ranking

Reishi Yokomori [†] Harvey Siy [††] Masami Noro [†] Katsuro Inoue [†††]
[†] Department of Software Engineering, Nanzan University
27 Seirei-cho, Seto, Aichi 489-0863, Japan
[††] Department of Computer Science, University of Nebraska at Omaha
6001 Dodge Street, Omaha, NE 68182, USA
[†††] Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
{yokomori, yoshie}@se.nanzan-u.ac.jp, hsiy@mail.unomaha.edu, inoue@ist.osaka-u.ac.jp

Abstract

Most of today's software applications are built on top of libraries or frameworks. Just as applications evolve, libraries and frameworks also evolve. Upgrading is straightforward when the framework changes preserve the API and behavior of the offered services. However, in most cases, major changes are introduced with the new framework release, which can have a significant impact on the application. Hence, a common question a framework user might ask is, "Is it worth upgrading to the new framework version?" In this paper, we study the evolution of an application and its underlying framework to understand the information we can get through a multi-version use relation analysis. We use component rank changes to measure this impact. Component rank measurement is a way of quantifying the importance of a component by its usage. As framework components are used by applications, the rankings of the components are changed. We use component ranking to identify the core components in each framework version. We also confirm that upgrading to the new framework version has an impact to a component rank of the entire system and the framework, and this impact not only involves components which use the framework directly, but also other indirectly-related components. Finally, we also confirm that there is a difference in the growth of use relations between application and framework.

1. Introduction

In software development, creating applications on top of frameworks is a widely-accepted practice. Frameworks contain reusable functions, models and code patterns. Developers build on the framework with their own functions, reducing development interval while maintaining software quality.

Modern software systems often consist of hundreds or thousands of classes, packages, functions and modules. Important information is scattered all over the software system. Analysis based on use-relations between components is essential to grasp the entire picture. However, use relation analysis is a very effort-intensive task due to the need to perform syntactic and semantic analysis of source code. Most of the previous work on use-relation analysis has focused on analyzing individual software versions rather than multiple versions. We suggested a method for detecting important updates in a development history by analyzing use relations[14]. In [14], we suggested a metric which represents the overall impact of the update by calculating the difference between two component ranks, before and after update. However, we also have to analyze inside the software system by using use relations and metrics calculated from them.

In this paper, we analyze the evolution of the use relationship between the framework and an application built on it. We make use of several metrics calculated from use relations. In an experiment, each version of the combined software system is analyzed first, and then each version of the application and the associated framework version are analyzed separately. We extract external use relations between framework and application. Based on these use relations, we analyze the system by using several metrics such as number of incoming and outgoing edges, component rank[8], and the impact of the update. In the experiment, trends of the above metrics of each subsystem are also analyzed and we weigh the differences between the evolution of distribution of incoming and outgoing edges.

The impact of upgrading to a new framework version is analyzed by investigating how component ranks change as a result of the update. By determining the effect of framework updates in the ranks and the growth of use relations, application developers can recognize where to pay attention when they update the framework. The sensitivity of the component rank metric is also evaluated by comparing the component ranks of the framework classes with and without the application classes. By finding functions and functional groups used in actual applications, framework developers can organize their framework to simplify its usage.

Through these analyses of use relations across multiple versions in a software repository, we observe the effects such external use relations would have on both the framework and the application, and consider characteristics of a development project which uses a framework. These analyses provide a rich source of information, alerting us to the fact that we can observe the growth of software multilaterally by also using the growth of use relations.

In Section 2, we present models and approach based on use relations as background. The results of applying the approach to an open source project are presented in Section 3. Finally, we discuss the effectiveness of the model and related works in Section 4.

2. Background

2.1. Component Graph

In general, a component is a modular part of a system that encapsulates its content and whose manifestation is replaceable within its environment [9, 11].

We model software systems by using a weighted directed graph, called a *Component Graph*[8]. A node in the graph represents a software component[1], and a directed edge from node x to y represents a *use relation* meaning that component x uses component y. If the graph consists of several software systems, a node may be combined with another node in certain conditions, such as because names of these nodes are the same, or contents of these nodes are almost same. Figure 1 shows an example of component graph for software system X. X consists of 5 components A – E. This graph also shows that component C uses both A and B, and D and E use C.

By using a component graph, we can easily identify the use relations between components and count the incoming and outgoing edges of a component. And we also calculate a component rank by using the component graph.

2.2. Component Rank Model

Based on the concept of the component graph, we proposed a component rank model. This rank is determined by

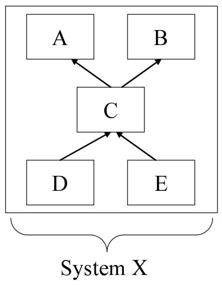

Figure 1. Component Graph

the ordering of values in an eigenvector for an adjacency matrix, which is derived from a given component graph.

Intuitively, we regard the given graph as a Markov chain[1]. If the chain is irreducible, a calculation of steady-state distribution always converges without recourse to an initial condition. So the initial values of nodes are quite same values, and then the steady-state distribution of the graph is calculated by power method on the adjacency matrix. Each value of the nodes is the value in steady-state distribution, in other words, the value in the eigenvector for the maximal eigenvalue of the matrix. Components are sorted by the value of the nodes, and we call the ranked data, the *component rank* of a set of components. For details, please refer to [8].

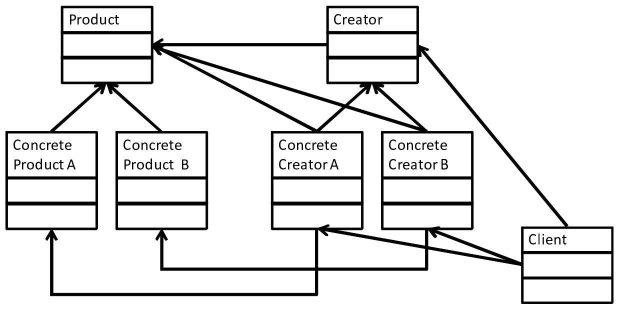

Figure 2. Component Graph for a Factory Method pattern

For example, a system which realizes a factory method pattern[4] is represented as a component graph in Figure 2. Table 1 is a component rank for this system[2]. Product and Creator are used by many other classes, so these classes are determined as important from the ranking. Thus, the more a component is directly or indirectly used, the higher is its rank. Components ranked high are generally important data structures or will play an important role in understanding

[1]"Component" is used in the broad sense of the word to mean any class, function, package, etc., that represents a logical unit in the program.

[2]This component rank takes into consideration pseudo use-relations to converge[8].

the system's behavior.

Table 1. Component rank for Figure2

Rank	Class	value
1	Product	0.40931
2	Creator	0.14301
3	ConcreteProductA	0.09699
3	ConcreteProductB	0.09699
5	ConcreteCreatorA	0.09128
5	ConcreteCreatorB	0.09128
7	Client	0.07113

Component rank is also used as a ranking method in SPARS-J [3], a search engine for Java software components. In [8], we indicate that component rank is useful for a component search engine through evaluation experiments, which showed that component rank moves the search result closer to the result expected by user.

2.3. An Update-Evaluation Metric based on Component Rank

As another usage of component rank model, we suggested a method for detecting important updates in a development history[14]. In the model, we considered that an update which brings sweeping changes in use relations is an important update because such update has a considerable impact on the entire system. We hypothesized that component rank also changes along with the change of use relations between components, and suggested a way to measure the use relation's change by calculating the difference in two component ranks, before and after update.

Based on the method, we developed a system which calculates the metric for each update, by obtaining source code from CVS and analyzing these code versions. We applied the system to several open source projects as an evaluation experiment. As a result, we can identify not only major-scale updates such as large function addition, but also relatively smaller updates, such as maintenance activities to core components, and refactoring and re-structuring of a software system. Some of these updates are not large in terms of lines of code (LOC) changed, but they are important updates because they have a large impact on the entire system. We confirmed that using the metric to quantify the use relation's change is effective to get another perspective on the source code's growth.

3. Experiment

In this paper, we analyze use relations between a framework library and an application which uses the framework.

[3]SPARS-J demo: http://demo.spars.info/

The component ranks and use relation metrics are calculated using SPARS-J, which treats Java classes as components. We consider the following as use relations: declaration of variables, creation of instances, method calls, and reference of fields. An analysis of use relations uses only static analysis, so dynamic binding is excluded.

We investigate the evolution of use relations over several releases of the application, determining whether use relations between framework and application show some trends, by using several metrics.

3.1. Preparation

In the experiment, we analyze the JHotDraw framework[4], a Java-based GUI framework for technical and structured graphics. As an application, we also analyze JARP[5]. JARP is a Java-based Petri tool using JHotDraw as a framework for editing a Petri net, drawing the result, and so on.

Both JARP and JHotDraw are hosted in SourceForge [6], from which we obtained the source code for each version of JARP and JHotDraw. JARP released 11 versions from 2001 to 2006, and we analyze these 11 versions along with the JHotDraw release used in each version.

Table 2 is a summary of 11 versions of JARP. Each version of JARP uses a certain version of JHotDraw, for example, JARP version 1.0.0 used JHotDraw 5.1, and JARP version 1.1.12 used JHotDraw 5.3, and so on. The class column refers to total number of classes, both of JARP classes and of JHotDraw classes. LOC is a fully inclusive sum, both JHotDraw and JARP, this set corresponds to Set 1 described below.

Table 2. The history of JARP

	Versions	Date	JHD	Class	LOC
1	1.0.0	2001/1/21	5.1	196(41+155)	23K
2	1.0.0.1	2001/1/26	5.1	196(41+155)	23K
3	1.0.1	2001/1/27	5.1	196(41+155)	23K
4	1.1.9	2001/4/30	5.1	284(129+155)	29K
5	1.1.10	2001/10/14	5.2	304(133+171)	31K
6	1.1.11	2001/11/1	5.2	312(141+171)	32K
7	1.1.12	2001/12/12	5.3	416(174+242)	42K
8	1.1.13	2003/4/22	5.3	433(191+242)	44K
9	1.1.14	2004/6/24	5.4	740(215+525)	82K
10	1.1.15	2005/2/11	5.4	740(215+525)	82K
11	1.1.16	2006/7/30	5.4	740(215+525)	82K

3.2. A Procedure of Analysis

In this experiment, we use only direct use relations, such as declaration of variables, creation of instances, method calls and reference of fields as use relations.

[4]JHotDraw: http://www.jhotdraw.org/

[5]JARP: http://sourceforge.net/projects/jarp/

[6]SourceForge.net - Open Source Software: http://sourceforge.net/

We extract the following metrics for each version:

(a) The number of outgoing edges of each node

This means how many classes the class uses. Even if there are more than one use relations between two classes, we treat as one use-relation exists between these classes.

(b) The number of incoming edges of each node

This means how many classes use this class. As in the case of outgoing edges, we exclude duplicate use relations.

(c) A value of each node in steady-state distribution

These values are calculated from a component graph which includes both JHotDraw and JARP, and are used for decision of component rank for a given component set (both JHotDraw and JARP, only JARP and only JHotDraw).

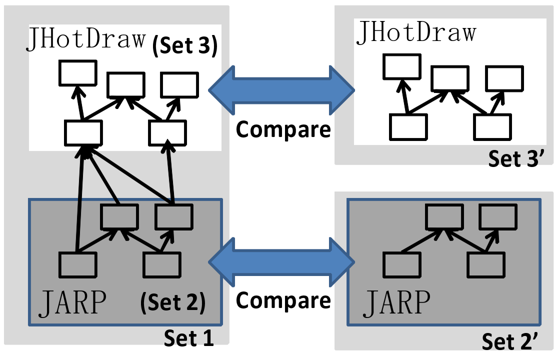

Figure 3. Analysis Overview

Figure 3 is an overview of the analysis. The procedure of analysis is the following:

(1) We analyze the entire of system (both JARP and JHotDraw) by using SPARS-J system. The analysis result is called Set 1. Set 1 is not used immediately, but is used for calculation of Set 2 and Set 3.

(2) We divide Set 1 into a result of classes which belong to JARP package (Set 2), and the one which belong to JHotDraw package (Set 3).

(3) We analyze JARP by itself by using SPARS-J system. The analysis result is called Set 2'. Set 2' considers only internal use relations in JARP.

(4) In the same way, we analyze JHotDraw by itself. The analysis result is called Set 3'. Set 3' considers only internal use relations in JHotDraw.

(5) By comparing these results, we calculate the number of external incoming edges and external outgoing edges which exist between JARP and JHotDraw. We also check the change of component rank and components whose metrics are sensibly changed.

3.3. Overall Results

We measure an update-evaluation metric, described in Section 2, with general metrics such as number of classes, LOC, and so on.

The metric represents the overall degree of changes in component rank values. In the metric, component rank is normalized into the value between 0 and 1, and metric calculates average of component rank's change for each component which exists both before and after update.

We obtain component rank for Set 1, Set 2 and Set 3 respectively, and calculate update-evaluation metrics among 11 versions. Table 3 is the result. For example, in the case of version 1.1.9, the result shows that the relative rank of each class moves up or down 6% on average. Considering only JARP classes, the component rank of each class moves up or down 26% on average. So we can consider that this version is a major update for JARP. Overall, updates for version 1.1.9, 1.1.12 and 1.1.14 increases the number of classes and changes its component rank substantially, and can be considered as major revisions for JARP.

Next, we focus attention on the change of component rank of subsystems (Sets 2 and 3). In the case of Set 2(JARP), versions 1.1.9, 1.1.12 and 1.1.14 change their component rank. These three updates change class allocation drastically, so many components moved to another package and JARP is re-organized. We can find the architecture restructured several times in these updates. For example, in the case of version 1.1.9, functions assumed by mainwindow class are divided into other subcomponents, and tools package is newly produced and many functions are implemented or moved to tools package. In addition, in version 1.1.12, functions assumed by PetriNetImpl are also divided into other subcomponents, and edition and simulation packages appear.

In the case of Set 3(JHotDraw), version 1.1.10, version 1.1.12 and version 1.1.14 have relatively high values. This is because the version of JHotDraw was updated in such releases. Specially, in the case of version 1.1.14, the number of classes in JHotDraw increases 290 classes with the introduction of JHotDraw version 5.4.

In the case of this system, we can confirm that component rank also changes substantially if the number of classes increases substantially. In this way, we can detect major updates by using the change of component rank, and we can also detect which subsystems are modified substantially by evaluating for each subsystem.

Table 3. Component rank update metrics

	Versions	All	JARP	JHotDraw
1	1.0.0	-	-	-
2	1.0.0.1	0	0	0
3	1.0.1	0	0	0
4	1.1.9	0.06	0.26	0.01
5	1.1.10	0.02	0.02	0.03
6	1.1.11	0.02	0.03	0.01
7	1.1.12	0.05	0.07	0.04
8	1.1.13	0.01	0.01	0
9	1.1.14	0.18	0.05	0.21
10	1.1.15	0	0	0
11	1.1.16	0	0	0

3.4. Analysis of Outgoing Edges

For each version, we count how many classes in JHotDraw each component in JARP uses. If a component uses other components several times, we treat as one use relation. We also find that JARP has some classes in JHotDraw with some modification; however, these classes are integrated into an original class during construction of component graph. Because there are some JHotDraw classes modified by JARP developers, use relations from JHotDraw to JARP may also exist. However, external use relations that we can confirm in 11 revisions are all from classes in JARP to classes in JHotDraw.

Table 4 is a summary of result on outgoing edges, with the package name of each component abbreviated. We extract 4 versions and show a top ten ranking for each version. For example, JDrawingView in version 1.0.0 uses 17 classes. It appears from the table that the outgoing edges per class is bounded (the values in the top rows are not steadily increasing).

To confirm that the maximum number of outgoing edges per class is bounded, we plot the cumulative frequency for each version in Figure 4. We can find that the number of classes which call the framework increases from one version to the next; however, within a given version, the number of class reaches a plateau early (most classes have less than 5 outgoing edges; beyond that, the cumulative number does not increase significantly).

To explain this observation, we focus on the evolution of components which use a lot of framework classes. In earlier versions, such components play multiple roles that it controls and implements an ambiguous and large feature. In later versions, implemented codes in such components are split into fragments of components.

We can picture a situation where a developer divides a large class into several small classes when a size of core class becomes too big to manage. In such a situation, usage of the framework is also divided into these small classes. So such component only controls the ambiguous and large feature, and uses only the essentials. In the latter period, classes for implementation and for handler, and so on use many framework components.

In the development of JARP, developers managed components not by designing precisely from the beginning, but by breaking down components that are too big. We think this is a better way for open source project because developers don't know which subsystem they should focus and develop with consideration of extensibility at the beginning of development.

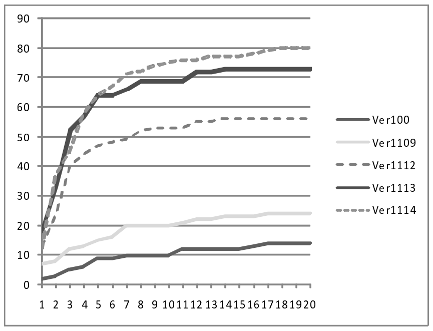

Figure 4. Cumulative frequency of outgoing external edge

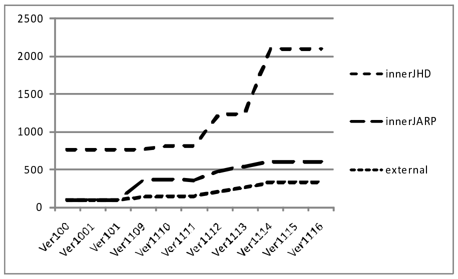

Figure 5. Growth of use relations

Figure 5 shows the growth of use relations over time. The number of internal use relations in JHotDraw and JARP, and as well as the external use relations from JARP to JHotDraw are counted. Most of the time, the internal use relations in JARP and the external use relations have increased by similar rates. In version 1.1.9, however, only the inner use relations in JARP increased, this is because JARP was restructured into a model-view-controller architecture.

3.5. Analysis of Incoming Edges

As in the case of outgoing edges, we also count how many JARP classes use each JHotDraw class. Table 5 is a

Table 4. Ranking of outgoing external use-relation (from JARP classes)

	Ver 1.0.0		Ver 1.1.9		Ver 1.1.12		Ver 1.1.14	
1	JDrawingView	17	JDrawingView	17	PlaceImpl	14	PlaceImpl	18
2	MainWindow	16	PetriPlaceImpl	14	TransitionImpl	12	ArcImpl	17
3	PetriPlaceImpl	11	PetriTransitionImpl	12	SelectionToolEx	12	TransitionImpl	16
4	PetriTransitionImpl	11	PetriSelectionTool	11	ArcImpl	9	DrawingPreview	13
5	PetriConnectionHandle	7	PetriNetImpl	7	PetriConnectionHandle	8	BendpointHandle	11
6	PetriSelectionTool	5	PetriConnectionHandle	7	PasteCommand	8	JDrawingView	10
7	PetriArcImpl	5	PetriDragTracker	7	CreationTool	8	WeightHandle	9
8	PetriSimulationTool	5	EditionTool	7	MainWindow	7	TokensHandle	9
9	JHDLoadTool	4	PetriArcImpl	6	JDrawingView	6	PasteCommand	8
10	PetriDragTracker	3	PetriSimulationTool	5	PetriNetImpl	5	PetriNetImpl	7

Table 5. Ranking of incoming external use-relation (to JHotDraw classes)

	JH 5.1 in JARP 1.0.0		JH 5.1 in JARP 1.1.9		JH 5.3 in JARP 1.1.12		JH 5.4 in JARP 1.1.14	
1	**DrawingView**	8	**DrawingView**	12	**Drawing**	16	**Figure**	26
2	**Drawing**	6	**DrawingEditor**	12	util.UndoableCommand	15	**FigureAttributeConstant**	25
3	util.StorableInput	5	**Drawing**	9	**DrawingEditor**	15	util.Command	23
4	**Figure**	5	**Figure**	8	**DrawingView**	12	**DrawingView**	21
5	util.StorableOutput	4	util.StorableOutput	4	**Figure**	10	**DrawingEditor**	19
6	**Tool**	4	util.StorableInput	4	util.StandardStorageFormat	7	util.UndoableCommand	17
7	**DrawingEditor**	4	**Tool**	4	util.Command	7	**Drawing**	17
8	**ConnectionFigure**	3	standard.AbstractFigure	4	**Tool**	6	**FigureEnumeration**	10
9	**Connector**	3	figures.AttributeFigure	3	standard.AlignCommand	6	util.StandardStorageFormat	9
10	standard.AbstractFigure	3	figures.TextFigure	3	standard.StandardDrawingView	6	**Tool**	8

summary of result about external incoming edges. We also extract 4 versions and show a top ten ranking for each version, with package names abbreviated. For example, DrawingView in version 1.0.0 was used by 8 classes in JARP.

Classes written in boldface came from the framework package in JHotDraw. The top ten classes are almost all in the framework package. Others are classes for dealing a figure and utility classes for command, input and output.

Unlike the case of outgoing edges, we observe that the increase in maximum number of incoming edge per class over the release history is open-ended. Figure 6 represents a cumulative frequency for each version. We find that the number of framework classes used by the application gradually increases over time, and the point at which we find the maximum number is getting larger for each version.

Usually, when developers need to use a feature from a framework class, they don't consider the number of times it is already in use. When development undergoes evolution and many features are implemented to the systems, such framework classes are used by many application classes, so such framework classes gradually become core components in the system. On the other hand, classes which have interfaces in the framework do not increase so much as long as the framework itself is not expanded or re-organized.

3.6. Analysis of Component Rank (JARP)

We extract JARP components from the overall system, and compute component ranks for JARP classes, corresponding to Set 2 in Figure 3.

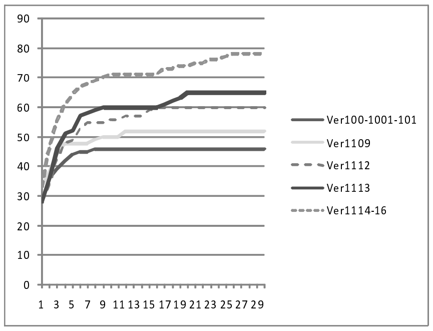

Figure 6. Cumulative frequency of incoming external edge

Table 6 shows the top ten classes ordered by component rank, for 4 JARP versions. In the early period, the top ten classes are almost about implementation class of Petri net. For example, the 1st to 6th, 8th and 9th classes in version 1.0.0 are all in a Petri package. However, in version 1.1.9, a lot of classes are added to the system, such as GUI classes (1st, 2nd, 5th and 8th), tool classes for XML browser (6th and 10th), and utility classes for configure and name treatment (3rd and 4th), so several kinds of classes appear in the list. The ranks of Petri net classes move down, however, they gradually move up again in version 1.1.12 and 1.1.14.

We can guess that features are added into such component also in latter period.

Component rank of application classes are affected by implementation of features. By referencing a ranking and checking a component whose rank moves up drastically, we can guess what kind of implementation is done.

In this development period, JARP developers upgraded the JHotDraw framework version three times, as shown in Table 2. To investigate the impact of framework updates on the application, we compared JARP component ranks before and after each framework update. Table 8, Table 9 and Table 10 are lists of JARP components which significantly improved in component rank after these framework updates. Components which were deleted or added in the updated framework version are taken into account in the calculation of component rank, however, these components are excluded from the result for comparison.

These results shows that many tool classes appear in every list. Main and utility classes, such as Hash, Main, and SplashWindow, also appear in the lists. By upgrading the framework, developers change components which access the framework, and main and utility classes also had to be adapted. So, we must pay attention not only components which uses the framework directly, but also main and utility classes when framework classes are upgraded.

Table 8. JARP Component rank's change between ver1.1.9 and 1.1.10 (129 components)

	Class	ver9	ver10	diff
1	PetriNetImpl	92	62	30
2	Crc32Hash	71	59	12
3	Hash	24	17	7
3	PetriSelectionTool	80	73	7
5	25 components	-	-	1

Table 9. JARP Component rank's change between ver1.1.11 and 1.1.12 (124 components)

	Class	ver11	ver12	diff
1	PNMLStorageFormat	107	59	48
2	FormatTool	94	54	40
3	AlignTool	94	55	39
4	EditionTool	94	60	34
5	Main	60	30	30
5	LoadTool	94	64	30
7	PrintTool	94	78	16
8	FileFilterImpl	107	93	14
9	SplashWindow	78	66	12
10	PetriNetMarking	21	13	8

Table 10. JARP Component rank's change between ver1.1.13 and 1.1.14 (130 components)

	Class	ver13	ver14	diff
1	LanguageTool	107	22	85
2	ChangeNetNameTool	117	84	33
2	FindPathAnalysis	117	84	33
4	SelectionTool	104	72	32
5	Main	31	6	25
6	PetriInvariantAnalysis	77	53	24
7	CommentTool	107	84	23
7	GridTool	107	84	23
7	NewTool	107	84	23
7	NewWindowTool	107	84	23

3.7. Analysis of Component Rank (JHotDraw)

As in the case of JARP, we also compute component ranks for JHotDraw classes, corresponding to Set 3 in Figure 3.

The summary is in Table 7. Classes in boldface come from the framework package in JHotDraw. Through the progression of JARP versions, highly ranked components seldom move dramatically. These classes are almost all in the framework package, except for a few in util. Some classes in framework package are consistently high; such classes are recognized as core components in the framework. Some classes, such as Storable, Connector, Handle, FigureChangeEvent and so on, are used only a few times by JARP or not at all, but are ranked high. These are heavily used internally by other JHotDraw classes.

In the result of Table 7, use relations from JARP are taken into consideration. It is noted that both JARP versions 1.0.0 and 1.1.9 use JHotDraw version 5.1, but the differences between the two JARP versions were not enough to affect the rank order of the top 10 JHotDraw components. To further assess the sensitivity of JHotDraw component ranks to the framework's usage by application classes, we also compute component ranks of JHotDraw by using only internal use relations (Set 3'). The component rank is almost same as the former one with respect to highly ranked components. This result shows that component rank of framework is mostly determined by the structure of the framework; the application largely does not impact the component rank measurements of the framework.

However, some lower-ranked components are affected by the use relations from the application. Table 11 and 12 are lists of JHotDraw components whose ranks improved drastically in version 1.1.9 and version 1.1.12 respectively. For example, AlignCommand in version 1.1.9 moved up 38 ranks, from 133 to 95, by taking into consideration the lone use relation from JARP.

The rightmost column represents the number of classes in JARP that use the component. Components used by application frequently do not appear so much because such

Table 6. Component Rank of JARP

	Ver 1.0.0	Ver 1.1.9	Ver 1.1.12	Ver 1.1.14
1	PetriNet	FindFilter	FindFilter	PetriNet
2	PetriNetEditor	FindProgressCallback	FindProgressCallback	FindFilter
3	PetriNetComponent	Config	Config	FindProgressCallback
4	PetriTransition	Name	Name	PetriNetEditor
5	PetriArc	EFileChooser	PetriNet	Tool
6	PetriPlace	XmlBrowser	PetriNetEditor	XMLResourceBundle
7	IntHashtableEntry	PetriNet	EFileChooser	ToolFactory
8	MainWindow	FindAccessory	XmlBrowser	figures.Transition
9	PetriStatesEnumAnalysis	PetriNetEditor	AbstractJARPTool	Main
10	ImageEncoder	AdapterNode	FindAccessory	figures.Place

Table 7. Component Rank of JHotDraw

	JH 5.1 in JARP 1.0.0	JH 5.1 in JARP 1.1.9	JH 5.3 in JARP 1.1.12	JH 5.4 in JARP 1.1.14
1	**Figure**	**Figure**	**Figure**	**Figure**
2	util.Storable	util.Storable	**FigureEnumeration**	**DrawingView**
3	**Connector**	**Connector**	**Connector**	**FigureEnumeration**
4	**FigureEnumeration**	**FigureEnumeration**	**Locator**	**JHotDrawRuntimeException**
5	**Locator**	**Locator**	**FigureChangeEvent**	**Connector**
6	**FigureChangeEvent**	**FigureChangeEvent**	**FigureChangeListener**	**ConnectionFigure**
7	**FigureChangeListener**	**FigureChangeListener**	util.Storable	**Drawing**
8	util.StorableInput	util.StorableInput	**DrawingView**	**Handle**
9	util.StorableOutput	util.StorableOutput	**ConnectionFigure**	util.CollectionsFactory
10	**ConnectionFigure**	**ConnectionFigure**	util.StorableInput	**DrawingEditor**

components have been already ranked high by inner use relation. However, as in the case of UndoableCommand in version 1.1.12, the rank of such component is drastically changed if the component is not used in the framework.

Other classes are not frequently used by application. These classes are not closely related to core components in the framework, and perform specific functions used only once in the application, such as command. Some classes, such as handler or connector classes, are affected by indirect use relations. Such components are also used to support design patterns.

4. Discussion

4.1. Significance of Use Relation Analysis

By analyzing changes in use relations, we can determine which updates are major updates. In this experiment, we can identify them through a change of number of classes and LOC, however, in general, some important updates are not big in terms of changed LOC, such as maintenance activities to core components, refactoring, re-structuring of a software system, and so on.

The number of incoming edges is also a good metric for understanding changes. However, if a framework function is divided into sub-functions and implemented in several classes in the latter periods, the result may only provide partial information. On the other hand, an evaluation of component rank takes into account indirect use relations, so we can identify components which support designing, such as handler and connector and so on.

From the analysis result, we can find that application classes repeat growth and breakdown cycles in the development. At first, implementation and control of a function are implemented to a class at the same time, and the class becomes bigger as one grows. However, when a size of the class is too big, implementation is divided into smaller classes. Component rank of application classes is affected by an addition of function, so we can roughly estimate the content of updates by checking a component whose rank is significantly changed.

Ichii investigated a distribution of a number of incoming edges and outgoing edges of component graph for a variety of software systems[12]. He reported the distribution of a number of outgoing edges is bounded by a size of class description, however, that of incoming edges is open-ended. Our analysis result fits well with his result.

In this experiment, some framework classes maintain a high component rank over time, so we can easily identify core components and core functions used by application classes. On the other hand, a few components are not in framework package and are not used in the framework

Table 11. JHotDraw Component rank's change in JARP ver1.1.9 (156 components)

	Class	Set3	Set3'	diff	Used
1	AlignCommand	95	133	38	1
1	ToggleGridCommand	95	133	38	1
3	ChangeAttributeCommand	92	125	33	1
3	DeleteCommand	75	108	33	2
5	DragTracker	81	111	30	1
6	ConnectionHandle	67	87	20	1
7	BringToFrontCommand	114	133	19	1
7	SendToBackCommand	114	133	19	1
9	BufferedUpdateStrategy	91	108	17	1
10	CopyCommand	119	133	14	1
10	CutCommand	119	133	14	1
10	PasteCommand	119	133	14	1
13	ChopEllipseConnector	77	88	11	1
14	GroupHandle	98	107	9	0
15	PolyLineHandle	52	60	8	1
15	SelectionTool	63	71	8	2
17	Clipboard	50	56	6	1
18	RadiusHandle	93	98	5	0
18	ShortestDistanceConnector	93	98	5	0
18	DrawingEditor	13	18	5	12

Table 12. JHotDraw Component rank's change in JARP ver1.1.12 (241 components)

	Class	Set3	Set3'	diff	Used
1	UndoableCommand	43	199	156	15
2	StorageFormatManager	48	201	153	4
3	AlignCommand	68	201	133	6
4	ChangeAttributeCommand	90	185	95	3
5	StandardDrawingView	61	147	86	6
6	UndoableTool	79	164	85	4
7	ToggleGridCommand	120	201	81	1
8	BringToFrontCommand	124	201	77	1
8	SendToBackCommand	124	201	77	1
10	RedoCommand	132	201	69	1
10	UndoCommand	132	201	69	1
12	DeleteCommand	102	167	65	1
13	CopyCommand	138	201	63	1
13	CutCommand	138	201	63	1
15	PasteCommand	159	201	42	1
16	UndoActivity	118	157	39	1
17	Alignment	55	89	34	6
18	StandardStorageFormat	49	79	30	7
19	ConnectionHandle	104	122	18	1
19	Clipboard	73	91	18	3

classes, but its rank drastically changed because of use relations from application classes. In a practical sense, such components might be included into the framework package because they can be considered as core components since they are used as frequently as the components in the framework. This kind of usage information represents what functions and functional groups are used in actual applications. Such knowledge can be used by framework developers in designing future enhancements to the framework that would simplify its usage.

4.2. Implications for Framework Upgrade

The number of incoming edges to classes in the framework almost has not been reduced through the development period as shown in Figure 5, so we can presume that existing APIs also have been firmly maintained, even as developers add new functions and APIs to the framework. In those cases, use relations from the application depend on a relatively small set of framework classes, though the number of relations per class may increase without bound. If the number of framework interfaces is tightly controlled, it is generally worth upgrading to a new framework version, unless existing APIs change. However, other factors such as quality and underlying defects should be taken into account in the decision to upgrade.

In the case where component ranks in a new framework version dramatically change, we can identify which application classes will need closer attention and testing by locating the affected framework classes and identifying their dependent application classes through the incoming edges. By knowing which application classes will likely be heavily impacted, the application developers can make a rough assessment of the work needed to upgrade to a new framework.

A core framework class may also be divided into several small classes in the process of framework evolution. So we can consider a situation where existing APIs are redesigned and their usages are completely changed. In such a situation, we can assume that a number of incoming edges to some core components in the framework decreases, as in the case of core application classes. We cannot find any such situation in this experiment; however, we will explore such situations in further replications of this study.

4.3. Related Works

Previous research on analysis of software repositories have focused on understanding reasons of software changes[5], identifying how communication delay among developers have effects on software development[7], detecting potential software changes and incomplete changes[15], and so on. In [10], Johnson proposed an approach that automatically records developer's activities with the objective of finding a relation between the internal characteristics (size and time, etc.) and the external characteristics (quality and reliability of products, etc.) rather than measuring updates.

Several researchers have also examined issues of framework evolution. Most of these studies focused on how framework or library developers can make it easier for user applications to migrate to newer versions of the framework by providing automated or semi-automated support, such as capturing and replaying API refactorings[6], creating compatibility layers[3], and providing annotations to generate and guide updates to applications[2, 13]. Our work complements these researches by focusing on the application developers and providing them with metrics-based guidance for deciding when to update to a new library version.

Our method is an extension of [14], and our goal is to understand how software evolves by analyzing use relations between software components. In this paper, we show that we can grasp further particulars by splitting the system being analyzed, e.g., into framework and application.

5. Conclusion

In this paper, we analyzed changes in use relations between framework and application, by using several metrics. We found that framework and application have several unique features in the growth of use relation. Component rank of application classes were affected by implementation of features, and some framework classes drastically improved in rank due to use relations from application classes. This kind of information represents what function and functional group are used in actual application and is useful for guiding future enhancements or redesigns of the framework.

As future work, we are planning to apply our method to other software systems to verify the external validity of our observations, and to refine the evaluation method based on how use relations evolved. Our ultimate target is establishment of the technique that can point out application components which should be modified or carefully inspected, and can roughly estimate the cost of upgrading to a new framework version.

Acknowledgments

This work was supported by KAKENHI (19700033), Grant-in-Aid for Young Scientists (B).

References

[1] G. Blom, L. Holst, and D. Sandell. *"Problems and snapshots from the world of probability"*. Springer, 1994.

[2] K. Chow and D. Notkin. "Semi-automatic update of applications in response to library changes". In *Proceedings of the International Conference on Software Maintenance (ICSM '96)*, pages 359–368, Monterey, California, 1996.

[3] D. Dig, S. Negara, V. Mohindra, and R. Johnson. "ReBA: refactoring-aware binary adaptation of evolving libraries". In *Proceedings of the International Conference on Software Engineering (ICSE '08)*, pages 441–450, Leipzig, Germany, 2008.

[4] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *"Design Patterns: Elements of Reusable Object-Oriented Software"*. Addison Wesley, 1995.

[5] D. German and A.Mockus. "Automating the measurement of open source projects". In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, pages 63–67, Portland, Oregon, 2003.

[6] J. Henkel and A. Diwan. "CatchUp!: capturing and replaying refactorings to support API evolution". In *Proceedings of the International Conference on Software Engineering (ICSE '05)*, pages 274–283, St. Louis, Missouri, 2005.

[7] J. D. Herbsleb, A. Mockus, T. A. Finholt, and R. E. Grinter. "An empirical study of global software development: Distance and speed ". In *Proceedings of the 23rd international conference on Software Engineering*, pages 81–90, Toronto, Canada, 2001.

[8] 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.

[9] I. Jacobson, M. Griss, and P. Jonsson. *"Software Reuse"*. Addison Wesley, 1997.

[10] P. M. Johnson, H. Kou, J. M. Agustin, Q. Zhang, A. Kagawa, and T. Yamashita. "Practical automated process and product metric collection and analysis in a classroom setting: lessons learned from Hackystat- UH". In *Proceedings of the 2004 intl. Symposium on Empirical Software Engineering (ISESE2004)*, pages 136–144, Redondo beach, CA, 2004.

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

[12] M.Ichii, M. Matsusita, and K. Inoue. "An Exploration of Power-Law in Use-Relation of Java Software Systems". In *Proceedings of the 19th Australian Conference on Software Engineering*, pages 422–431, Perth, WA, Australia, May 2008.

[13] J. H. Perkins. "Automatically generating refactorings to support API evolution". In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE '05)*, pages 111–114, Lisbon, Portugal, 2005.

[14] R. Yokomori, M. Noro, and K. Inoue. "Evaluation of Source Code Updates in Software Development Based on Component Rank". In *Proceedings of 13th Asia Pacific Software Engineering Conference*, pages 327–334, Bangalore, India, 2006.

[15] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. "Mining version histories to guide software changes". In *Proceedings of the 26th international conference on Software Engineering*, pages 563–572, Edinburgh, Scotland, 2004.