Evolution of Component Relationships between Framework and Application

Reishi Yokomori[1] Harvey Siy[2] Norihiro Yoshida[3] Masami Noro[1] Katsuro Inoue[4]

[1] Department of Software Engineering, Nanzan University
27 Seirei-cho, Seto, Aichi 489-0863, Japan
{yokomori, yoshie}@se.nanzan-u.ac.jp

[2] Department of Computer Science, University of Nebraska at Omaha
6001 Dodge Street, Omaha, NE 68182, USA
hsiy@mail.unomaha.edu

[3] Graduate School of Information Science, Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0192, Japan
yoshida@is.naist.jp

[4] Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
inoue@ist.osaka-u.ac.jp

Received 10 January 2012; Revised 21 May 2012; Accepted 25 May 2012

Abstract. Most of today's software applications are built on top of libraries or frameworks. The increasing number of cloud-based services gives rise to 3rd party frameworks that offer such services from a cloud platform. Just as applications evolve, frameworks also evolve. Such evolution is even more pronounced in frameworks that underlie cloud-based services. Upgrading is straightforward when the framework changes preserve the API and behavior of the offered services. However, major changes are introduced with the new framework release, which have a significant impact on the application. A framework user has to consider how to adjust to the new version. In this paper, we study the evolution of an application and its underlying framework through a multi-version analysis. For the analysis, we investigate two kinds of component relationships: one is component rank, the other is clone relation. 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 confirm that upgrading to the new framework version has an impact to a component rank of the entire system. On the other hand, existence of code clone shows how application developers use existing framework code as a reference, and removal of clones shows which reuse activities were recognized as problematic. Analysis of results from these relationships provides useful insights into developers' activities.

Keywords: software evolution, component relationships, use-relation, clone-relation

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. With the rise of cloud computing, such frameworks can now be hosted on cloud-based platforms, further expanding the number of potential applications using them. The expectations for frequent and reliable upgrades in cloud-based services [1] makes it all the more important to understand the coevolution of applications and underlying frameworks.

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 relations between components is essential to grasp the entire picture. However, relation analysis is a very effort-intensive task due to the need to perform several kinds of source code analyses. Most of the previous work 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 [2]. In [2], we suggested a metric which represents the overall impact of the update by calculating the difference between two component ranks, before and after update. How-

ever, we also have to analyze inside the software system by studying how several kinds of component relationships change over time.

In this paper, we analyze the evolution of the relationship between a framework and an application built on it. While the analysis is based on a traditional framework that resides locally, the methodology can be applied to any component-based software system whether the framework is on the same host or hosted in a cloud.

During the analysis, we calculate two kinds of metrics from component relationship; one is metrics based on use relations, and the other is the one based on clone relations. For the metrics based on use relations, 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 also 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 [3], 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.

Next, for the metrics based on clone relations, we identify application code components that are clones of framework components, and follow the change in these relationships over multiple versions. When we checked development history, some code clones are produced in particular version while other code clones disappear. By checking such kinds of production and extinction, we can understand how a framework user uses existing codes as a reference and which part was recognized as a problem.

Through these analyses of relations across multiple versions in a software repository, we observe 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 component relations.

This paper is a revised version of [4]. For this paper, we add an analysis based on clone relation and discussion new findings. In Section 2, we present models and approach based on component 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 [5],[6]. We model software systems by using a weighted directed graph, called a **Component Graph** [3]. 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*. 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.

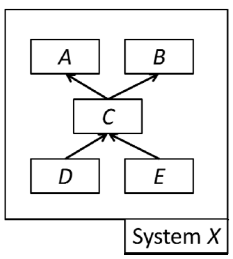

Fig.1. An Example of Component Graph

2.2 Component Rank Model

Based on the concept of the component graph, we proposed a component rank model. This rank is determined by 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 [7]. 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 [3].

For example, a system which realizes a factory method pattern [8] is represented as a component graph in Figure 2. Table 1 is a component rank for this system. **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 the system's behavior.

Component rank is also used as a ranking method in SPARS-J, a search engine for Java software components. In [3], 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.

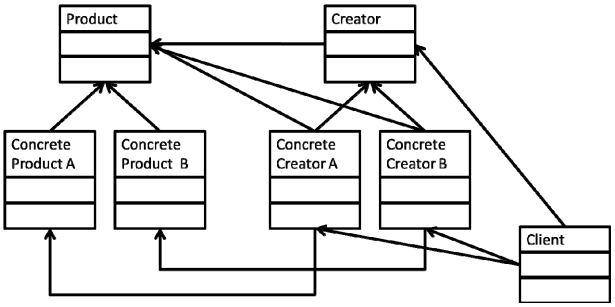

Fig.2. Component Graph for a Factory Method pattern

Table 1 Component rank for Fig. 2.

Rank	Class name	Evaluation 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

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 [2]. 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.

2.4 Component Graph based on Clone Relations

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 [9],[10],[11]. 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. A pair of such similar code fragments is often called **clone pair** because both of them have to be modified similarly and at a same time in many cases and developers have to grasp semantic cohesiveness of them. Clone relation metrics are calculated using CCFinder [11]. 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.

3 Experiment

In this paper, we analyze use relations between a framework library and an application which uses the framework, and clone relations in the application related with the framework library. We investigate the evolution of these two kinds of relations over several releases of the application, determining whether relations show some trends, by using several metrics.

For use relation, 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. The component ranks and use relation metrics are calculated using SPARS-J, which treats Java classes as components.

For clone relation, we extract application side's code clones that use framework API or contribute an execution of framework API, such as pre-processing code or post-processing code, from the analysis result of CCFinder. Some of such code clones exist between (or among) application components and others exist between application component and framework component. We will represent these relations over several releases by using graphs and explain these relations.

3.1 Preparation

In the experiment, we analyze the JHotDraw framework, a Java-based GUI framework for technical and structured graphics. As an application, we also analyze JARP. 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, 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 Use relation 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. 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 of JHotDraw and JARP, only JARP and only JHotDraw).

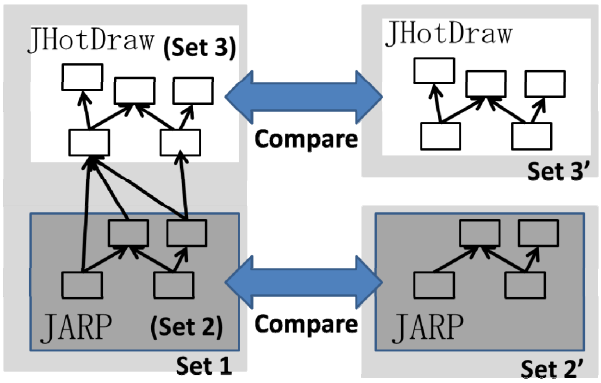

Fig.3. Overview of use relation analysis

Figure 3 is an overview of the use relation analysis. The procedure of the 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 A Procedure of Clone relation Analysis

In the case of Clone relation, we have to distinguish whether extracted clone relations are related with use of framework or not. For this experiment, this work is done manually because it is difficult to do correct judgments. Some clone relations are recognized as clone because tokens in a series of method-call statements matches each other contingently, or tokens in repeated computations for different purpose fit each other by chance. On the flip side, preprocessing codes for framework sometimes have no direct use relation to framework classes; however, the part should be recognized as code clone.

So we manually inspected all clones extracted by CCFinder, and present these relations by graph visualization. In the experiment, clone relations represented in each Figure are significant clones. For example, code fragments recognized as clone pair are a whole of method related with framework, a series of member definitions for similar purposes, pre or post processing codes for use of framework, a series of code fragment that uses the framework API, and so on. Some clone relations lies astride application class and framework class, we also show it on the graph.

3.4 Results about Update Evaluation Metric

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

Table 3 Update metrics based on Component rank

	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

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.

3.5 Results based on Use Relation Analysis: About 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).

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

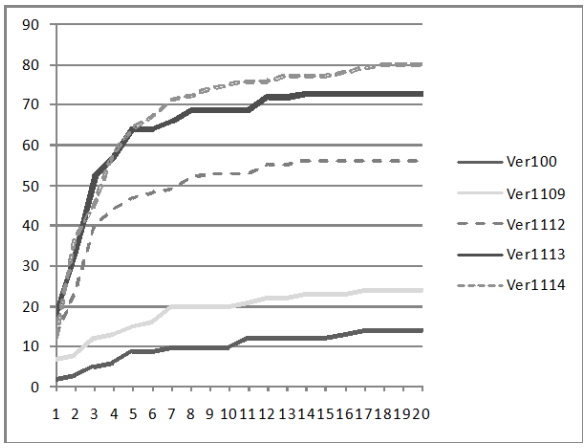

Fig.4. Cumulative frequency of outgoing external edge

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.

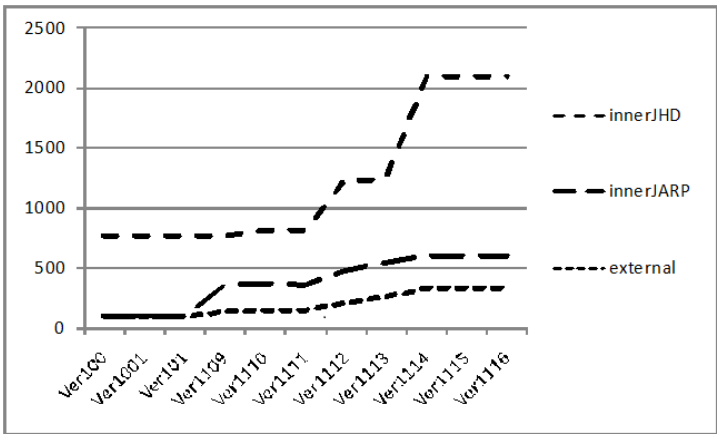

Fig.5. Growth of use relations

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, howev-

er, only the inner use relations in JARP increased, this is because JARP was restructured into a model-view-controller architecture.

3.6 Results based on Use Relation Analysis: About Incoming edges

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

Table 5 Ranking of incoming external use-relation (from JARP classes)

	Ver 1.0.0		*Ver 1.1.9*		*Ver 1.1.12*		*Ver 1.1.14*	
1	**DrawingView**	8	**DrawingView**	12	**Drawing**	16	**Figure**	26
2	**Drawing**	6	**DrawingEditor**	12	UndoableCommand	15	**FigureAttributeConstant**	25
3	StorableInput	5	**Drawing**	9	**DrawingEditor**	15	Command	23
4	**Figure**	5	**Figure**	8	**DrawingView**	12	**DrawingView**	21
5	StorableOutput	4	StorableOutput	4	**Figure**	10	**DrawingEditor**	19
6	**Tool**	4	StorableInput	4	StandardStorageFormat	7	UndoableCommand	17
7	**DrawingEditor**	4	**Tool**	4	Command	7	**Drawing**	17
8	**ConnectionFigure**	3	AbstractFigure	4	**Tool**	6	**FigureEnumeration**	10
9	**Connector**	3	AttributeFigure	3	AlignCommand	6	StandardStorageFormat	9
10	AbstractFigure	3	TextFigure	3	StandardDrawingView	6	**Tool**	8

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 is not expanded or re-organized.

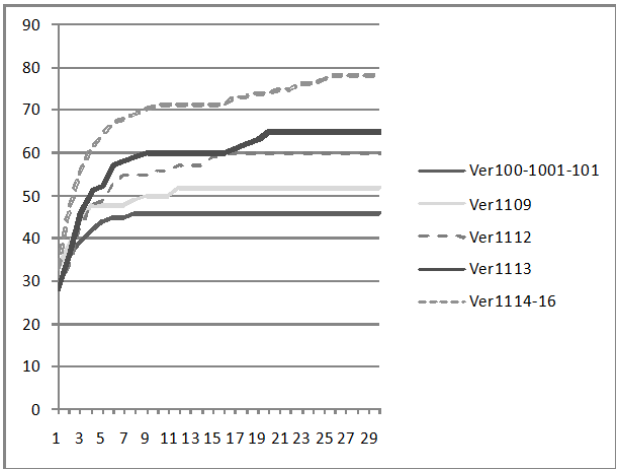

Fig.6. Cumulative frequency of incoming external edge

3.7 Results based on Use Relation Analysis: About Component Rank for JARP Components

We extract JARP components from the overall system, and compute component ranks for JARP classes, corresponding to Set 2 in Figure 3. 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 7, 8 and 9 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 show 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 6 Component Rank for JARP Components

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

	Class	Ver.9	Ver.10	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 8 JARP Component rank's change between ver1.1.11 and 1.1.12 (124 components)

	Class	Ver.11	Ver.12	Diff
1	PNMLStorageFor-mat	107	59	48
2	FormatTool	94	54	40
3	AlignTool	94	55	39
4	EditionTool	94	60	34
5	Main	60	30	30
6	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 9 JARP Component rank's change between ver1.1.13 and 1.1.14 (130 components)

	Class	Ver.13	Ver.14	Diff
1	PNMLStorageFor-mat	107	22	85
2	FormatTool	117	84	33
3	AlignTool	117	84	33
4	EditionTool	104	72	32
5	Main	31	6	25
6	LoadTool	77	53	24
7	PrintTool	107	84	23
8	FileFilterImpl	107	84	23
9	SplashWindow	107	84	23
10	PetriNetMarking	107	84	23

3.8 Results based on Use Relation Analysis: About Component Rank for JHotDraw Components

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 10. 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.

Table 10 Component Rank for JHotDraw Components

	JH5.1 in JARP 1.0.0	*JH5.1in JARP 1.1.9*	*JH5.3 in JARP 1.1.12*	*JH5.4 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**	**JHotDraw RuntimeException**
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**

In the result of Table 10, 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 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.

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

	Class	Set3	Set3'	Diff	Used		Class	Set3	Set3'	Diff	Used
1	AlignCommand	95	133	38	1	**10**	CutCommand	119	133	14	1
1	ToggleGridCommand	95	133	38	1	**10**	PasteCommand	119	133	14	1
3	ChangeAttribute Command	92	125	33	1	**13**	ChopEllipse Connector	77	88	11	1
3	DeleteCommand	75	108	33	2	**14**	GroupHandle	98	107	9	0
5	DragTracker	81	111	30	1	**15**	PolyLineHandle	52	60	8	1
6	ConnectionHandle	67	87	20	1	**15**	SelectionTool	63	71	8	2
7	BringToFrontCommand	114	133	19	1	**17**	Clipboard	50	56	6	1
7	SendToBackCommand	114	133	19	1	**18**	RadiusHandle	93	98	5	0
9	BufferedUpdateStrate-gy	91	108	17	1	**18**	ShortestDistance Connector	93	98	5	0
10	CopyCommand	119	133	14	1	**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		Class	Set3	Set3'	Diff	Used
1	UndoableCommand	43	199	156	15	**10**	UndoCommand	132	201	69	1
2	StorageFormatManager	48	201	153	4	**12**	DeleteCommand	102	167	65	1
3	AlignCommand	68	201	153	6	**13**	CopyCommand	138	201	63	1
4	ChangeAttribute Command	90	185	95	3	**13**	CutCommand	138	201	63	1
5	StandardDrawingView	61	147	86	6	**15**	PasteCommand	159	201	42	1
6	UndoableTool	79	164	85	4	**16**	UndoActivity	118	157	39	1
7	ToggleGridCommand	120	201	81	1	**17**	Alignment	55	89	34	6
8	BringToFrontCommand	124	201	77	1	**18**	StandardStorageFor-mat	49	79	30	7
8	SendToBackCommand	124	201	77	1	**19**	ConnectionHandle	104	122	18	1
10	RedoCommand	132	201	69	1	**19**	Clipboard	73	91	18	3

3.9 Results based on Clone Relation Analysis

Next, we will analyze the growth of framework-related clone relations between JARP and JHotDraw. At first, we show a summary of the framework-related clone relations of each 11 versions at Table 13. Each column represents a number of clone relations between JARP classes and the one between JARP class and JHotDraw class. In the case of the metric, the presence of a group of strongly connected classes yields high value; For example, in version 1.1.13, there is a group which consisted of 6 strongly connected classes and the group yielded 15 clone relations. This Table shows that clone relation also grows up as the system evolves. Furthermore, we recognized that developers have tried clone elimination in parallel with producing clone relations.

Table 13 A Growth of Framework-Related Clone relations

	Versions	Clone Relation in JARP	Clone Relation in JARP and JHotDraw
1	1.0.0	1	3
2	1.0.0.1	1	3
3	1.0.1	1	3
4	1.1.9	2	3
5	1.1.10	5	6
6	1.1.11	10	6
7	1.1.12	19	2
8	1.1.13	27	4
9	1.1.14	11	16
10	1.1.15	11	16
11	1.1.16	11	16

From here, we will show a graph that represents two kind of framework-related clone relations. The one is clone relations between JARP classes; those are represented in left side of each figure, and the other is the ones between JARP class and JHotDraw class; those are represented by across a center vertical line in each figure.

Ver.1.0.0, Ver.1.0.0.1 and Ver.1.0.1. The first three versions have the same class structure and clone relations, which are shown in the left box of Figure 7. We see two clone groups that mean more than three classes are connected strongly; one associated with **NodeFigure** and the other with **StandardDrawingView**. The clone related to **NodeFigure** includes two entire methods, findConnector() and drawConnectors(), part of features of **NodeFigure.** These were introduced from JHotDraw to JARP. The clone relations were dissolved in the next version. The clone related to **StandardDrawingView** includes 21 methods and almost the entire class was introduced from JHotDraw to JARP. This clone relation is kept till version 1.1.10, and **JDrawingView** is modified in time with the modification of JHotDraw in the version 1.1.10. **JDrawingView** becomes a subclass of **StandardDrawingView** from the version 1.1.11. This clone relation was suddenly dissolved in version 1.1.12; however, it returned in version 1.1.13 because several method descriptions were added in this version.

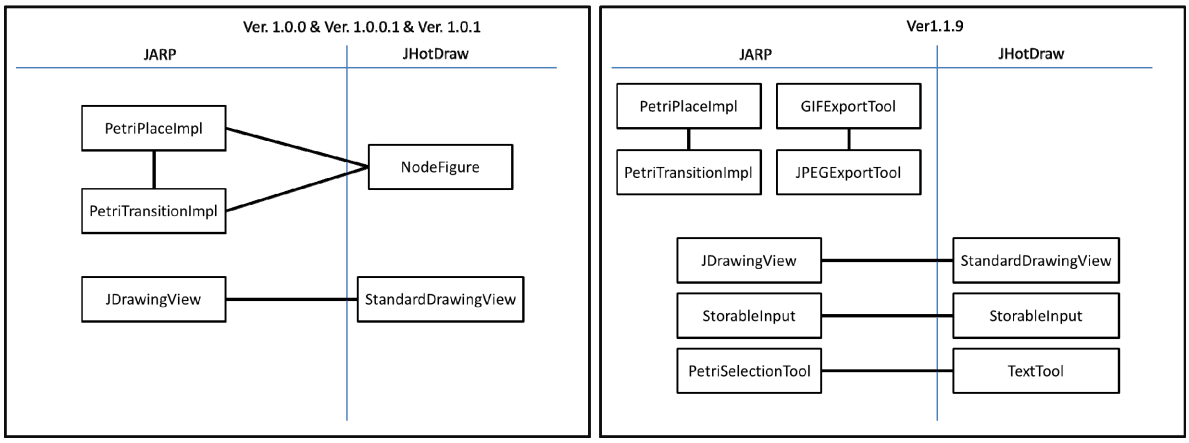

Fig.7. Clone relations related with Framework (a)

Ver.1.1.9. The right box of Figure 7 shows the clone relations of version 1.1.9. We see 5 clone pairs, including 3 new clone pairs. The clone relation between **PetriPlaceImpl** and **PetriTransitionImpl** is kept from the previous version; however, the content of clone is quite different. These two classes were reorganized and these classes have a very similar structure in this version. Constructor and several common methods are recognized as clone, and this clone relation is kept throughout all the subsequent versions.

New clone relations are related to export tools, **StorableInput** and **TextTool,** respectively. **GIFExportTool** and **JPEGExportTool** have common preprocessing code in execute() and these are recognized as code clone. In the later version this relation spreads to classes about storage format. **StorableInput** in JHotDraw is customized for JARP, so several principal methods, such as readStorable() and makeInstance(), are the same. This clone relation is also kept throughout all subsequent versions. **PetriSelectionTool** imports features in **TextTool**, so we can find several common methods in these classes. However, features in **PetriSelectionTool** gradually become specialized features, so this clone relation is gradually diminished and disappears in version 1.1.12.

Ver.1.1.10. The left box of Figure 8 shows the clone relations of version 1.1.10. We can see 11 clone pairs, including 2 new clone groups. Clone relations related to export tools have spread to **SaveTool**. Clone code between **NewTool** and **LoadTool** are common pre-processing code in execute() and code treating Swing GUI. These classes were reorganized sometime in later development, so the clone relation was reduced to just having constructor code in common in version 1.1.12 and finally disappeared in version 1.1.13. In JHotDraw 5.2, several connection-related classes have a similar method (findConnectableFigure()), **PetriConnectionHandle** in JARP was implemented in a manner similar to **ConnectionHandle**-related classes in JHotDraw, so the method is recognized as code clone. These relations disappeared after JHotDraw was upgraded to JHotDraw 5.3.

Ver.1.1.11. The right box of Figure 8 shows the clone relations of version 1.1.11. We can see 16 clone pairs, including 2 new ones and another clone group which changed form. Created clones are the one between **PNMLStorageFormat** and **JPNStorageFormat**. These classes have a common restore() method, however, this was dissolved in the next version. The other created clone code is between **PetriSimulationTool** and **PetriSelectionTool**. JARP developers had started to reorganize structures of **Tool** related classes, and these classes became subclasses of **AbstractAction**, having a lot of common or similar methods including constructor, event handling methods, GUI handling methods and so on. This relation was dissolved in the next version.
Clone relations related to export tools changed into clone relations between several storage format classes. These classes handled different image file formats, however, they have similar code in store() method. Code that perform individual processing for each class are recognized as code clones, so these relations are difficult to be dissolved; these relations are kept to the end of our analysis period.

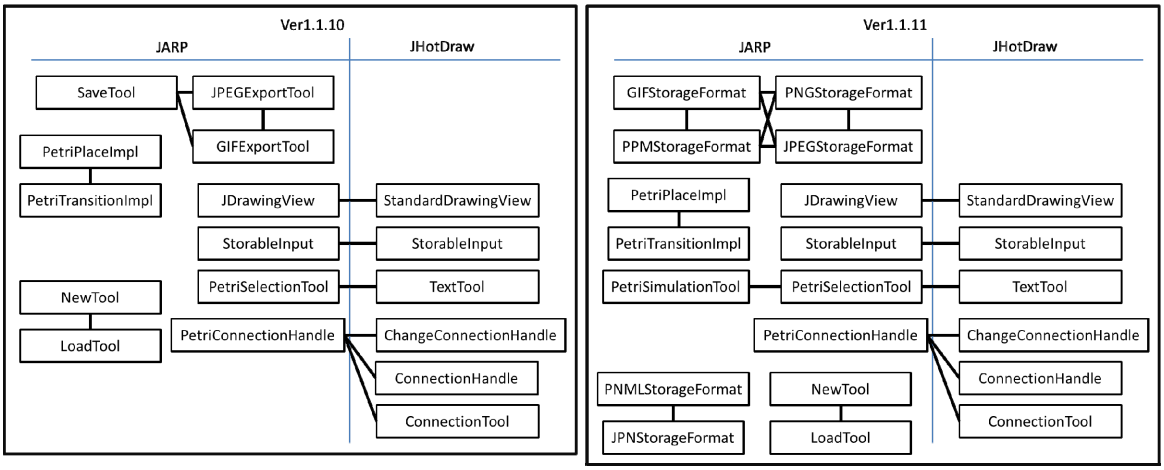

Fig.8. Clone relations related with Framework (b)

Ver.1.1.12. The left box of Figure 9 shows the clone relations of version 1.1.12, and we can find 21 clone pairs. Some components lost Petri- prefix from their name, so **PetriTransitionImpl** becomes **TransitionImpl** and **PetriPlaceImpl** becomes **PlaceImpl**, respectively. In this version, the class structure of JARP was reorganized, and almost half of clones are dissolved and new clone pairs appear in this version. Created clones are mainly related to tool classes in JARP and most of such classes are subclasses of **AbstractJARPTool**. In these classes, we can find a lot of stylized codes in their constructor and common methods. These clone relations spread till version 1.1.13. Both JARP and JHotDraw have a class (**CreationTool**) with the same name, however, they only have one method createFigure() in common.

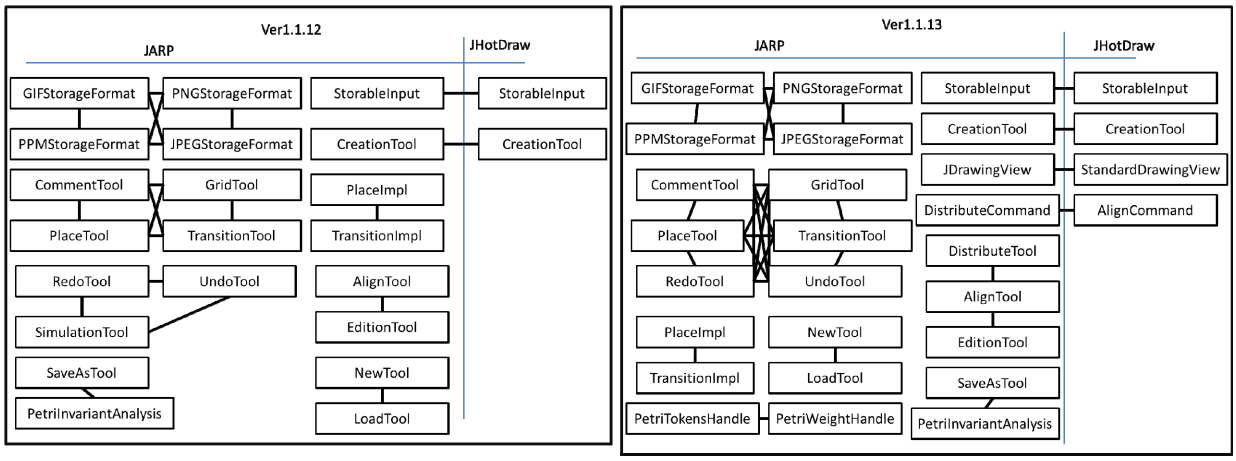

Fig.9. Clone relations related with Framework (c)

Ver.1.1.13. The right box of Figure 9 shows the clone relations of version 1.1.13. We can find 31 clone relations in this version. Two clone groups in the previous version joined together and became one large strongly connected clone group. This clone group is dissolved in the next version. **PetriTokensHandle** and **PetriWeightHandle** are subclasses of **LocatorHandle**, and they have a lot of common methods and constructors. **DistributeCommand** in JARP has an inner class named **UndoActivity** that realizes undo-related features and we can consider that these features are introduced through **AlignCommand** or other Command classes in JHotDraw. These relations are kept to the end of our analysis period.

Ver.1.1.14-16. The last three versions have the same class structure and their clone relations are quite the same, as shown in Figure 10. We find 27 clone relations in these versions, and we find 2 new large clone groups. One is between **BendpointHandle** and Tool classes in JHotDraw Version 5.4, and the other is between **CreateBendpointHandle** and other Tool classes in JHotDraw Version 5.4. These two clone groups have different kinds of clone code; the former have clone code in the description about undo feature, and the latter have clone code in draw() method and other related methods. **BendpointHandle** and **CreateBendpointHandle** also have code clones in similar operations, such as joinAndRemove(), splitAndMove(), and so on. **SaveAsTool** and **SaveTool** have similar operations in saveFile(), so this is also recognized as code clone.

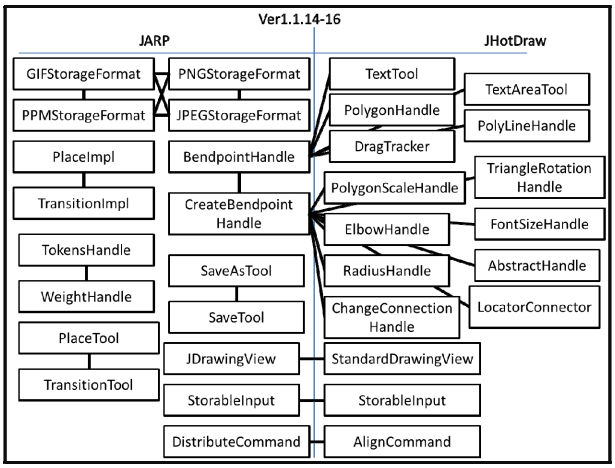

Fig.10. Clone relations related with Framework (d)

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 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 re-designed 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.

From a view point of code clone, application developer sometimes introduces executable codes from codes in library, and it becomes a code clone between framework and library. When the framework is upgraded, we have to be careful whether framework code to be copied is modified or not. If the original code in framework is modified, application developer also have to apply the same kind of modification to the destination code. We find almost the same situation at **JDrawingView** class in version 1.1.10, and it seems to be a frequent practice in the actual development. In [13], Kasper mentioned that disadvantage of API-derived clones is that modification of copied codes becomes more difficult when the original API evolves, and our experiments and discussion also arrive at a similar awareness.

4.3 Growth of code clones between framework and application

We checked the growth of framework-related clone in Section 3.9, and we consider that there are roughly several general phases of a framework-related clone's lifecycle. At first, application developer introduces code in framework into application to implement or use necessary features in framework. Such code represents how to implement framework features in the application and this activity produces clone relation between application and framework. If such code doesn't spread into application, this clone relation would be neglected because we cannot find any clone relations inside of application. Of course, application developers have to check the code when framework is upgraded.

We can also imagine a situation that a similar solution is adapted in another component for realizing similar features, i.e., a copy-and-paste reuse attempt. In such case, such code spreads into application code as a result; these become clone relations in application and framework. This situation would be least desirable, and these kinds of clones should be removed as soon as possible. If developers recognize such situation, they would try to reorganize class structure to dissolve them. This would be the second phase, however, we cannot find a good example to explain it in our analysis. This is because all versions we studied are released products, and such clones would be dissolved prior to the release of a product.

The third phase is clones in reorganized class structure. After reorganization of class structure, we can find a lot of stylized codes in methods and constructor. These clones would be less harmful than clones of previous phase and they are manageable clones because developer can understand relationships between components by checking inheritance relations, and so on. And these clones are easy to dissolve by applying typical OO refactoring approaches to them. We hypothesize that adequate class structure is organized by passing through these phases. The generality of this hypothesis will have to be confirmed by analysis of additional software systems.

4.4 Automated Support for Framework Upgrades

Our existing analysis process is semi-automated. We apply SPARS-J and CC-finder to the target software system, and then results about clones are refined manually. After that, use and clone relations for each version are organized manually, which takes significant effort. We see several scenarios where comprehensive automated support for upgrading applications to newer framework releases would be desirable.

In the development of software based on cloud computing, the cloud-based frameworks could be upgraded at any time. Thus applications that use such frameworks have no control over whether they should upgrade to a new release or not. Nonetheless, these applications may behave differently after an upgrade. These applications may benefit from past experience in coping with similar framework changes, so sharing knowledge and information obtained from past development is important.

In other situations where application developers do have control over when to perform framework upgrade, it is complicated in general because they have to investigate which part of the framework is replaced on the revised version, and whether the changes affect the application or not. Costs for these activities may more expensive than expected because application developers also have to understand knowledge and information obtained from past framework development, if need arises.

In many case, an update history of the software in version control system is instrumental for such understanding. Developers produce change logs of the target component, and describe how the target component had been modified. This information is good for understanding about individual component; however, such information sometimes may be too fine to understand a whole picture. In such case, we have to understand relationships between related components, and draw the big picture mentally.

Our analysis gives developers several useful information pieces such as framework classes with significant changes in component rank and clone relationships, and can help developers make informed choices. We are planning to develop a tool that shows such information, and such tool would help tasks that require a plenty of time and toil. By visualizing the change in relationships before and after updates, the tool user can obtain a brief summary of the update that is easier to understand. And by showing a summary of each update that includes changes in component rank, we can distinguish the important update among a lot of updates. Finally, by showing a change in clone relationship, the tool can show where redesign is needed and show that such redesign went well.

We also consider that the tool may support change impact analysis. By traversing edges that represent use relations in a component graph, we can identify a set of application classes that uses framework classes directly or indirectly, and are potentially impacted by the upgrade. This makes it easier for application developers to see the effort needed to upgrade to the new framework.

These aforementioned tools contribute to automating the application upgrade activity. While we recognize that a fully automated analysis environment is important for our methods, there are challenges for full automation: refinement of relation (especially clone relationship) and tracking of components that contains same features before and after update.

We use CC-finder to perform clone relation analysis. CC-finder's analysis is based on kind of tokens that appears in source code, so it has enough scalability for our purpose. However, raw analysis data produced by CC-finder sometimes includes clone pair that seems to be unfit to be called a clone pair; for example, a series of several method-call statements for different purpose is not adequate for code clones. However, alignment sequence of tokens of such code fragment is easy to match because CC-finder recognizes such code fragments as sequences of method call and parameter(s). These clones sometimes give wrong results, so we have to remove them automatically also in the automated tools. One of the powerful solutions is based on automatic removal of such clones by using particular metrics. In [14], Higo calls such code fragment language-dependent code clone, and suggests a methodology for removal based on metric, named RNR(S) that means a ratio of non-repeated code sequence in clone set S. By using such methods, our tool can show refined results for practical use.

Through our experiments, we confirmed that components grow gradually by feature additions through the incremental development, and finally they are decomposed into several components and each component is reorganized to moderate size. In such situation, there are a lot of components whose name or whose container package are changed. In addition, component's name is sometimes changed so as to fit with its current features. These changes about component names are impediment for analysis for multiple versions, because tool users expect continuity on such components for usability, and so we have to take a special care for such events. At the moment, we are considering a method based on metrics that are available in the current environment, such as a number of common methods and fields, or similarities between two components, and so on. However, there are a lot of studies in this research area [15] [16], so we have to study related works and establish specialized methods for our purpose by considering also analysis costs.

4.5 Related Works

Previous research on analysis of software repositories have focused on understanding reasons of software changes [17], identifying how communication delay among developers have effects on software development [18], detecting potential software changes and incomplete changes[19], and so on. In [20], 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. Tamura proposed a new approach to software reliability assessment based on deterministic chaos theory and confirmed its effectiveness by applying it to several development histories of open source software [21]. Black investigated fault data extracted from multiple versions of the open source system, by using two of Weiser's original slice-based metrics (Tightness and Overlap) [22].

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 refactoring [23], creating compatibility layers[24], and providing annotations to generate and guide updates to applications [25],[26]. 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 [2], 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. We also add an analysis based on evolution of clone relations.

From the viewpoint of use relation (component dependency) analysis, there is active research in architecture recovery. Zhang represents OO system by using WDCG (Weighted Directed Class Graph), and suggested a clustering algorithm for recovering high-level software architecture [27]. Constantinou represents hierarchical relationships between components as D-layer, by contracting closed paths in component graph, and investigated relationships between architecture layer and design metrics [28]. We consider that our contribution is applying dependency analysis to multi-version analysis.

From the viewpoint of clone analysis, Antoniol analyzed the evolution of code duplications in 19 versions of the Linux kernel [29]. Their target system seems to be relatively stabilized, so the evolution of common ratio is also stabilized through these versions. They also reported that recently-introduced architectures tend to exhibit a slightly higher cloning ratio. We consider that our target includes the early stage of incremental development, so the result shows another side of the clone evolutions. Mondal analyzed the stability of several kinds of cloned codes, and they reported that Type3 clones, known as gapped clones, have higher stability than other clones [30]. In [31], Yamanaka suggested a daily reporting system about modifications related with cloned codes. We consider that changes of component rank may also be fit to be used in such reporting system.

Kim surveyed matching techniques that can be used for multi-version program analyses and evaluate them based on hypothetical change scenarios [15]. Davies introduced a novel signature matching technique that uses a combination of metrics for tracing 3rd party components to their sources [16]. When we realize a support tool based on our findings, this survey and research would be helpful for establishing a matching policy between the versions before and after modification.

5 Conclusion

In this paper, we analyzed changes in use and clone 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. In the case of clone relations, we studied how clone relations evolved throughout the development. We found several clone relations between framework and application and framework-related clones among application components. From the result, we conjecture that there is a process from introducing features and related foundation in framework to organizing a clone-less class structure.

This work becomes more significant in the context of cloud computing where rapidly changing services are the norm and there is an expectation of reliable evolution or upgrading of services [1]. Recognizing patterns of use relationships can help hone the services offered by cloud-based frameworks. Such frameworks may have to balance the need for supporting existing users and the need to adjust in compliance with cloud service standards; so understanding component rankings gives framework maintainers the information needed to guide when, where and how to change the API.

In addition, there is an internal benefit for guiding framework maintainers in consolidating and refactoring existing services. Low component rankings may guide maintainers to houseclean unused services thus easing future maintenance efforts. Discovering clone relationships can also lead to identification and elimination of redundant services.

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 and clone relations evolved. We are also working to improve tool support as mentioned in Section 4.4. 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 from several viewpoints.

Acknowledgement

This research is supported by Nanzan University Pache Research Subsidy I-A-2 for the 2011 academic year.

References

[1] I. Neamtiu and T. Dumitras, "Cloud Software Upgrades: Challenges and Opportunities," in *Proceedings of the International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA' 11)*, 2011.

[2] R. Yokomori, M. Noro, K. Inoue, "Evaluation of Source Code Updates in Software Development Based on Component Rank," in *Proceedings of 13th Asia Pacific Software Engineering Conference*, pp. 327-334, 2006.

[3] K. Inoue and R. Yokomori, T. Yamamoto, M. Matsushita, S. Kusumoto, "Ranking Significance of Software Components Based on Use Relations," *IEEE Transactions on Software Engineering*, Vol. 31, No. 3, pp. 213-225, 2005.

[4] R. Yokomori, H. Siy, M. Noro, K. Inoue, "Assessing the Impact of Framework Changes Using Component Ranking," in *Proceedings of 25th IEEE International Conference on Software Maintenance*, pp. 189-198, 2009.

[5] I. Jacobson, M. Griss, P. Jonsson, *Software Reuse*, Addison-Wesley Professional, New York, 1997.

[6] C. Krueger, "Software Reuse," *ACM Computing Surveys*, Vol. 24, No. 2, pp. 131-183, 1992.

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

[8] E. Gamma, R. Helm, R. Johnson, J.M. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-wesley Professional, New York, 1995.

[9] L. Jiang, G. Misherghi, Z. Su, S. Glondu, "DECKARD: Scalable and Accurate Tree-based Detection of Code Clones," in *Proceedings of the 29th International Conference on Software Engineering*, pp. 96-105, 2007.

[10] Z. Li, S. Lu, S. Myagmar, Y. Zhou, "CP-miner: Finding Copy-paste and Related Bugs in Large-scale Software Code," *IEEE Transactions on Software Engineering*, Vol. 32, No. 3, pp. 176-192, 2006.

[11] T. Kamiya, S. Kusumoto, K. Inoue, "CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code," *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654-670, 2002.

[12] M. Ichii, M. Matsusita, K. Inoue, "An Exploration of Power-law in Use-relation of Java Software Systems," in *Proceedings of the 19th Australian Conference on Software Engineering*, pp. 422-431, 2008.

[13] C.J. Kapser and M.W. Godfrey, " "Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software," *Empirical Software Engineering*, Vol. 13, No. 6, pp. 645-692, 2008.

[14] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, "Method and Implementation for Investigating Code Clones in a Software System," *Information and Software Technology*, Vol. 49, No. 9-10, pp. 985-998, 2007.

[15] M. Kim and D. Notkin, "Program Element Matching for Multi-version Program Analyses," in *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR' 06)*, pp. 58-64, 2006.

[16] J. Davis, D.M. German, M.W. Godfrey, A. Hindle, "Software Bertillonage: Finding the Provenance of an Entity," in *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR' 11)*, pp. 183-192, 2011.

[17] D. German and A. Mockus, "Automating the Measurement of Open Source Projects," in *Proceedings of the 3rd Workshop on Open Source Software Engineering*, pp. 63-67, 2003.

[18] J.D. Herbsleb, A. Mockus, T.A. Finholt, R.E. Grinter, "An Empirical Study of Global Software Development: Distance and Speed," in *Proceedings of the 23rd International Conference on Software Engineering*, pp. 81-90, 2001.

[19] T. Zimmermann, P. Weissgerber, S. Diehl, A. Zeller, "Mining Version Histories to Guide Software Changes," in *Proceedings of the 26th International Conference on Software Engineering*, pp. 563-572, 2004.

[20] P.M. Johnson, H. Kou, J.M. Agustin, Q. Zhang, A. Kagawa, 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 International Symposium on Empirical Software Engineering (ISESE2004)*, pp. 136-144, 2004.

[21] Y. Tamura and S. Yamada, "A Method of Reliability Assessment Based on Deterministic Chaos Theory for an Open Source Software," in *Proceedings of the Second International Conference on Secure System Integration and Reliability Improvement (SSIRI '08)*, pp. 60-66, 2008.

[22] S. Black, S. Counsell, T. Hall, D. Biwes, "Fault Analysis in OSS Based on Program Slicing Metrics," in *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2009)*, pp. 3-10, 2009.

[23] 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)*, pp. 274-283, 2005.

[24] D. Dig, S. Negara, V. Mohindra, R. Johnson, "ReBA: Refactoring-aware Binary Adaptation of Evolving Libraries," in *Proceedings of the International Conference on Software Engineering (ICSE '08)*, pp. 441-450, 2008.

[25] 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)*, pp. 359-368, 1996.

[26] 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)*, pp. 111-114, 2005.

[27] Q. Zhangs, D. Qiu, Q. Tian, L. Sun, "Object-oriented Software Architecture Recovery Using A New Hybrid Clustering Algorithm," in *Proceedings of the Seventh International Conference on Fuzzy Systems and Knowledge Discovery*, Vol. 6, pp. 2546-2550, 2010.

[28] E. Constantinou, G. Kakarontzas, I. Stamelos, "Towards Open Source Software System Architecture Recovery Using Design Metrics," in *Proceedings of the 15th Panhellenic Conference on Informatics*, pp. 166-170, 2011.

[29] G. Antoniol, U. Villano, E. Merio, M. Di Penta, "Analyzing Cloning Evolution in the Linux Kernel," *Information and Software Technology*, Vol. 44, No. 13, pp. 755-765, 2002.

[30] M. Mondal, C.K. Roy, M.S. Rahman, R.K. Saha, J. Krinke, K.A. Schneider, "Comparative Stability of Cloned and Non-cloned Code: An Empirical Study," in *Proceedings of the 27th ACM Symposium on Applied Computing*, 2012.

[31] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, T. Sano, "Industrial Application of Clone Change Management System," to be appeared in *Proceedings of the 5th International Workshop of Software Clones*, 2012.