Evaluation of Source Code Updates in Software Development Based on Component Rank

Reishi Yokomori † Masami Noro † Katsuro Inoue ††
† Department of Information and Telecommunication Engineering, Nanzan University
27 Seirei-cho, Seto, Aichi 489-0863, Japan
†† Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
{yokomori, yoshie}@it.nanzan-u.ac.jp, inoue@ist.osaka-u.ac.jp

Abstract

Essential activities for the achievement of trouble-free software development are monitoring a software product and management of a software project. Monitoring changes that have major impact, however, is usually a very hard to complete because every engineer usually could not know entire source code in detail. In this paper, we propose our metric for source code updates based on component rank. Software components and their use-relation alter as development goes. The component rank also changes as a response to the changes of use-relation among components. We use the degree representing change of component rank as a metric of impact for a source code update in a development. We applied the metric to open source projects, and demonstrated that the metric is useful to know refactoring activities or important updates. We also discuss how the metric can contribute for process management.

1. Introduction

Software used in computer systems for our daily life is getting larger and more complex. As its size grows more and more, greater number of software engineers must be engaged in development projects. In such environment, monitoring a project and providing feedback on the development are vital to promote a trouble-free development. Many projects use a configuration management system, such as CVS, for source code control. Developers make a lot of updates to the system in its development cycle. Source codes controlled in the configuration management system are kinds of ideal materials for analysis of important updates which affect the software system. Such information is a good predictor for the monitoring and the postmortem of the project.

However, to get entire understanding of stored source codes is a very arduous task and takes a plenty of time, so quantitative information based on a specialized metric is one of the most important key issues. The commonly used metric is LOC; however, to try to understand the progress of the development with just LOC may leads mischievous conclusion. LOC increases monotonically in many cases through the development process. The important updates, as a result, are often buried in the monotonous increase of LOC. We insist that a metric which can figure out an impact of update is also required for effective analysis.

In this paper, we suggest an update-evaluation model to detect important updates, which affect the whole software system, among a lot of updates. In the model we proposed, we use component rank suggested in [5, 6]. The component rank is a metric that evaluates importance of components based on use-relation among software components. Components used by many components and used by important components are evaluated as important components. In the model, a component rank is calculated at each time when source code is updated. A set of component rank shows a transition on importance of each component. In development phase, there are many updates, such as modifications, function-additions, refactoring components, and so on. These updates set off changes on use-relation among components. If we focus attention on each component, the rank of the component also changes as a response to the modification to its use-relation.

We make a hypothesis that an important update changes use relation drastically and also induces the fluctuation of the component rank. So we define the degree of change of each component rank as a quantified measure of impact about source code update. We applied the metric to several open source projects in sourceforge[12]. Throughout the experiment, we confirmed that the metric is useful for capturing updates about core components and system architec-

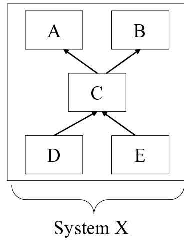

Figure 1. An example of component graph

ture, such as refactoring, rewriting, and so forth. The metric can provide useful information when we evaluate updates occurred in a development.

In Section 2, we present a component rank model suggested in [5, 6]. Section 3 shows an update-evaluation model based on the component rank and implementation of the model. The results of applications for open source projects will be presented in Section 4. Finally, we discuss about the effectiveness of the model in Section 5.

2. Component Rank Model

Our method is based on component rank model suggested in [5, 6]. In this section, we present the model and elements used in the model.

2.1. Component Graph

In component rank model, software systems are modeled by a weighted directed graph, called a *Component Graph*. A node in the graph represents a software component, and a directed edge e_{xy} from node x to y represents a *use relation* meaning that component x uses component y. In current implementation, it defines that components are Java classes, and use relations are the class inheritance, method invocation, abstract class implementation, and so on. However, the model does not restrict to a specific kind of component and use relation.

Figure 1 shows a 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. A software system is generally modeled with a component graph that is weakly connected (assuming that there is no redundant component). Our evaluation model targets at the graph which means one software system.

2.2. Weight of Node

Component rank model defines each weight of node in the graph based on the following computation policies. This model calls the order of the nodes sorted by the weights component rank of the components. This model introduces several definitions to define $w(v)$ for component graph $G = (V, E)$.

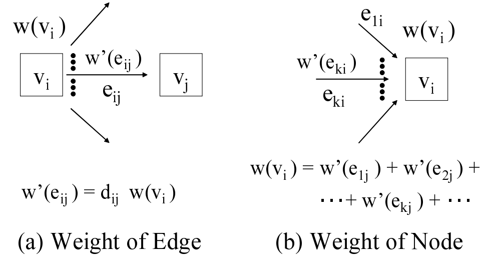

Figure 2. Definition of weights

Definition 1 (Total Weights of Nodes) *Each node v in component graph G has a non-negative weight value $w(v)$ where $0 \leq w(v) \leq 1$. For simplicity of following calculation, it assumes that the sum of the weights of all nodes in G is 1, i.e., $\sum_{v \in V} w(v) = 1$.*

Definition 2 (Weight of Edge) *For computation of the weights of nodes, it introduces the weight $w'(e_{ij})$ of an edge $e_{ij} = (v_i, v_j)$, such that $w'(e_{ij}) = d_{ij} \times w(v_i)$.*

Figure 2 (a) depicts this definition. Here, d_{ij} is called a *distribution ratio*, where $0 \leq d_{ij} \leq 1$ and the total of d_{ij} for each j is 1. If there is no edge from v_i to v_j, $d_{ij} = 0$. The distribution ratio d_{ij} is used for determining the forwarding weights of v_i to an adjacent node v_j. In the current implementation, the distribution ratios of the use relations, which emanate from one node, are equal values.

Definition 3 (Weight of Node) *The weight of a node v_i is defined as the sum of the weights of all incoming edges e_{ki}, such that $w(v_i) = \sum_{e_{ki} \in IN(v_i)} w'(e_{ki})$.*

Here, IN(v_i) is the set of the incoming edges of v_i. Figure 2 (b) shows this definition.

2.3. Computation of Weights

Based on these definitions, it has $n(= |V|)$ simultaneous equations for $w(v_i)$, such that

$w(v_i) = \sum_{e_{ki} \in IN(v_i)} d_{ki} \times w(v_k)$.

Assume that W is a vector of node's weights,

$$W = \begin{pmatrix} w(v_1) \\ w(v_2) \\ \vdots \\ w(v_n) \end{pmatrix}.$$

Also, D is a matrix of the distribution ratios,

$$D = \begin{pmatrix} d_{11} & d_{12} & \dots & d_{1n} \\ d_{21} & d_{22} & \dots & d_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ d_{n1} & d_{n2} & \dots & d_{nn} \end{pmatrix}.$$

So the simultaneous equations can be rewritten by,

$$W = D^t W \quad (1)$$

where D^t is the transposed matrix of D.

Together with Definition 1, formula (1) can be solved by computing the eigenvector with eigenvalue 1. Instead of computing the eigenvector, it can also compute the weights of each node by a repeated computation such that, it gives initial ad-hoc weights to each node(e.g., $1/n$ to each node), and then propagate them to adjacent nodes through directed edges. The weights are repeatedly recomputed until the all weights become stable. If we assume that movement of software developer's focus on the target components is represented by a probabilistic state transition, the component graph represents a Markov chain. Thus, computing the weights of the nodes in the graph corresponds to getting a stationary distribution of the chain.

Figure 3 shows a component graph with computed weights. v_1 has two outgoing edges, and weight 0.4 is evenly divided to two outgoing edges with 0.2 each (i.e., $d_{12} = d_{13}$ = 0.5). v_3 has two incoming edges, each with weight 0.2, so that the weight of v_3 is 0.4.

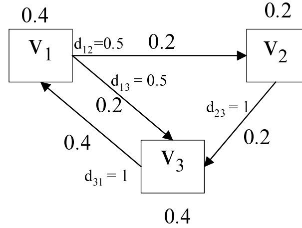

Figure 3. An example of stable weights assigned to nodes and edges

To hasten and ascertain a convergence of this computation, the model introduces pseudo use relation. It also introduces a concept of clustered component graph to track the copied and modified components among a lot of software systems. The component rank model was implemented as a part of component search engine, named *SPARS-J*[13], and its application result was reasonable such that very general and core classes are ranked high (significant), and specific and independent classes are ranked low (insignificant). The details are described in [5, 6].

3. An Update-Evaluation Metric based on Component Rank

When we have a time-line view of a software development process, a software system is gradually completed by implementing necessary functions, and data and control structures needed for them. We expect these implementations induce various modifications on components, and components are stabilized as the software system is near completion or reaches maturity. We can understand these modifications by reading source codes thoroughly, however it's hard for every engineer to do this. From a viewpoint of sharing information, evaluation based on a quantified measure of impact about source code update is a fair solution. The simplest way is based on increase and decrease of simple metrics, such as LOC, number of classes, and so on. However, these metrics increase monotonically through its development cycle, and they are susceptible to changes which are large-scale but not so important. So we think these metrics cannot provide enough information for understanding updates.

In this paper, we suggest an update-evaluation model based on component rank. Our proposed method is based on a hypothesis that the more system is modified massively, the more two component ranks before and after the update differs vastly. As a ground of this hypothesis, we consider the fact that an update that has a major impact on the whole system also yields a lot of modifications in use-relation among components.

For example, we consider the case that a function is added to a system. By adding a function, existing components, such as a library, and data structures, become used more frequently. In this case, the ranks of such components move up, and the rank of other components fall down in comparison. The larger the scale of the added function, the more the number of affected components. In another case, refactoring and rewriting activities are not uncommon because a structure of large software is very easy to get complex. We can also expect that these activities cause a far reaching impact on use-relation by extraction of methods and classes, and readjustment of interfaces. Such readjustments yield a component which drops out of use, or whose usage is quite changed. When these updates are performed, the rank of such components moves up or down significantly. Especially, its fluctuation becomes stronger when such modification extends to core components.

In proposed model, it calculates component rank at each time of source code update. The set of component ranks means a transition of importance of each component. To calculate an impact of a certain update, we extract two sets of components; a set of components before the update and the one after the update. For these two sets of components, we calculate two component ranks separately. These two

component ranks generally has a lot of common elements (continuing components), so we can calculate the degree of up and down of each continuing component's rank.

We evaluate an impact of the update, by calculating an average of each component's rank change. In the development phase, there are many updates, such as modifications, function-addition, refactoring, and so on. Proposed metric is affected by not only the mere scale of the change, but also the scale of the change at the view point of the use-relation, the whole scale of the software system and the degree of incidence for core components. By considering these factors, the proposed metric represents a more proper and quantified measure of the impact for updates.

3.1. Computational procedure

We introduce several definitions to define an impact $impCR(U)$ for a certain update U. A set of components before the update U is represented by $Set_{pre}(U)$, and the one after the update is represented by $Set_{aft}(U)$, respectively. We can also assume that there are corresponding components between $Set_{pre}(U)$ and $Set_{aft}(U)$, so define a set of corresponding components as

$$Set_{uni}(U) = \{X | X \in Set_{pre}(U) \land X \in Set_{aft}(U)\}.$$

And then, we can calculate two component ranks $CR_{pre}(U)$, and $CR_{aft}(U)$ for $Set_{pre}(U)$, $Set_{aft}(U)$, respectively. However, there are a lot of updates which adds new components or deletes the unneeded ones, so the number of components may increase or decrease. Therefore, we evaluate the impact of U by using the following normalized rank rather than using the raw rank. Each rank is normalized into a value between 0 and 1; the value 1 means the highest rank, 0 means the last place.

Definition 4 (Normalized Rank of Component X**)** *We assume that the rank of component* X *in* $Set_{pre}(U)$ $(Set_{aft}(U))$ *is* $CR_{pre}(U,X)$ $(CR_{aft}(U,X))$*, and that the number of components in* $Set_{pre}(U)(Set_{aft}(U))$ *is* $Num_{pre}(U)(Num_{aft}(U))$*.*
We define normalized rank for component X *as*

$$\begin{aligned} NCR_{pre}(U,X) &= \tfrac{Num_{pre}(U)-(CR_{pre}(U,X)-1)}{Num_{pre}(U)}, \\ NCR_{aft}(U,X) &= \tfrac{Num_{aft}(U)-(CR_{aft}(U,X)-1)}{Num_{aft}(U)}. \end{aligned}$$

Finally, we calculate a difference between two normalized ranks of each continuing component, and then calculate an average of the difference as $impCR(U)$. $ImpCR(U)$ means how much impact U gave to the component rank by a quantitative value.

Definition 5 (Impact of Update U **)** *We assume that the number of components in* $Set_{uni}(U)$ *is* $Num_{uni}(U)$*. We define an impact of* U*,* $impCR(U)$*, as*

$$impCR(U) = \frac{\sum_{X \in Set_{uni}(U)} |NCR_{aft}(U,X) - NCR_{pre}(U,X)|}{Num_{uni}(U)}.$$

3.2. Implementation of Evaluation Tool

We have designed and implemented a system to compute $impCR(U)$ for each update U in a development process. A target project is a project whose software is written in Java programming language, and whose source code is managed by CVS. Figure 4 shows an architecture of our first prototype of the system. This system uses a registration-subsystem of SPARS-J[6], so definitions about component is the same as the one of SPARS-J: Java class and interface source code are considered as a component, and use relations are class inheritance, interface implementation, abstract class implementation, variable declaration, instance creation, field access, and method invocation.

(1) Update-List-Generation phase:
The system obtains a list of update time and date from CVS by using a history command, and generates a list of update to be analyzed. In current implementation, we define a granularity of update as a day, so the system generates a list of update date.

(2) CR-Calculation phase:
Based on the list generated by phase (1), the system calculates a component rank for each update as following;

(2-1) The system obtains the update day's source codes by check-out command.

(2-2) The system calculates a component rank for source codes obtained by (2-1). In current implementation, it uses a registration-subsystem of SPARS-J.

(3) Metrics-Calculation phase:
Based on the component ranks calculated in phase (2), the system calculates an impact of changes for each update. For a visualization of the result to share information, we create a chart about $impCR$.

(3-1) For each update U, the system pulls out two sets of components correspond with $Set_{pre}(U)$ and $Set_{after}(U)$ and two component ranks correspond with $CR_{pre}(U,X)$ and $CR_{after}(U,X)$.

(3-2) For each update U, the system calculates $impCR(U)$ based on the definition.

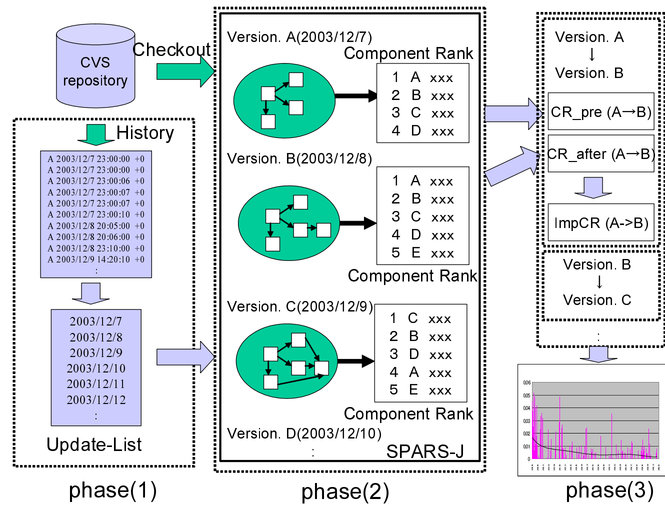

Figure 4. Architecture of the evaluation system

4. Experiment

In this section, we evaluate our approach by applying to several open source projects. In the experiment, to confirm to which kind of update the proposed metric yields a high value, we roughly follow up a content of the update which yields a high impCR value, from actual development history of the project.

4.1. Preparation

Sourceforge[12] is a very huge open source community and there are many development projects. In the experiment, we have extracted target projects that meet the following requirements from development projects in sourceforge; (1) written in Java, (2) CVS repository is used for source code control, (3) its history is stored in the CVS repository.

4.2. experiment description

We applied our system to 12 open source projects, presented in Table 1. We made a graph for each results, and investigated content of updates for every distinguishing point in every graph. These target projects vary in size and development period. Therefore, analysis times for these project are variously; the shortest one (galleon) is about 20 minutes (average 20 seconds per revision), and the longest one (jedit) is about 10 days (average 10 minutes per revision). At first, we explain the entire tendency of the application, and then show the concrete result of a project with several graphs.

Table 1. Target projects

	Name	From	To	Revision	LOC
1	azureus	2003/7	2006/2	903	374K
2	freemind	2000/8	2006/2	309	23K
3	galleon	2005/2	2006/2	55	91K
4	ganttproject	2003/5	2006/2	543	73K
5	jasperreports	2003/12	2006/2	349	149K
6	jbidwatcher	2000/5	2006/1	303	29K
7	jedit	2000/1	2006/2	1515	830K
8	megamek	2002/2	2006/2	990	107K
9	openwfe	2003/6	2006/2	643	81K
10	pmd	2002/6	2006/2	802	124K
11	pydev	2003/7	2006/2	315	45K
12	xui	2003/3	2006/2	68	153K

4.2.1 entire tendency

As overall tendency in all projects, we confirmed the fact that impCR decreases with progress of the development project. In the early stage of development, impCR indicates a high value because necessary features are implemented to an imperfect system one by one. On the other hand, at the time of nearly completion of the system, the impact of update becomes small because use-relation among existing components have already been stabilized.

We also confirmed that there is not so big difference through the development progress in respect of the maximum fluctuation range of component rank in a certain update. We think that this is because we defined a unit of update as a day, and this shows that the size of each update differs little through a development progress. On the other hand, entire size of the system (the amount of source

code) is getting larger through a development progress, so the size of update is relatively getting smaller; this is also a factor that impCR decreases.

We also applied polynomial approximation to a series of impCR for each project, and the result was a decreasing, convergent, and a little waving curve in many projects. We think that this is a characteristic of open source development that always adds features and releases a new version in a short cycle. A project whose products in early stage is not registered into CVS decreases the tendency of "decreasing" in the curve, and a project in which both an active period and a silent period are clearly appeared increases the tendency of "waving" in the curve.

In addition, we investigated contents of updates which scored high impCR value. A mere bug correction is hardly seen in them, and it is possible to classify those updates into the following three kinds roughly. By calculating and visualizing impCR, we think that we can extract only important updates which include a lot of modifications in use relation, and which have a huge impact on the system.

(1) Independent large-scale function implementation
This kind of updates add a few new core components and a lot of core data structures to the system. However, existing components are free of impact of the function implementation in many cases because of its independency. In the early stage of development, impact caused by adding core components is very huge, however, the impact is getting smaller as the progress of development. Especially, a large-scale but local update has this tendency strongly.

Example 1: In the case of TiVo Media server system galleon[1], updates which scored high impCR value are updates for correspondence to weather-information, music-player, online-radio, photo thumbnail, and iTunes etc. These individual functions have comparatively high independence, so each update's impact on the existing components is not so high. Therefore, impCR value of this kind of update has decreased gradually as development progress.

Example 2: JEdit[8] is a text editor that has a high extendibility by plug-in support. Therefore, a lot of additional functions are implemented as a lot of plug-ins. The independencies among such plug-ins are high, so impCR value of this kind of updates has decreased dominantly as the progress.

(2) Function implementation to core components and implementation of crosscutting features
In the case of implementing a new function to core components, impact caused by the update is very huge because new use-relations are added into the around of the modified core components. In the case of implementation of crosscutting features, such as a debugging feature, a logging feature, new data structures, and so on, impact caused by the update is also very huge because new use-relations are added into all around of the components in the system. This kind of update generally scores high impCR value and it is hard to be affected by the progress.

Example 3: PMD[11] is a detecting tool for potential problems which are liable to appear in Java source code. At the later stage of the development of PMD, updates which scored high impCR value are a function implementation to AST, correspondence to JAVA1.5 syntax, correspondence to JSP, and so on. In these updates, developers made a lot of modifications into a parsing subsystem, which includes many core components.

(3) Refactoring and rewriting
This kind of update has a huge impact on existing components because such updates includes a lot of movement of function(method) between components and a lot of modifications, deletion and removal to its data structure. Especially, the impact is very huge when restructuring of core components is performed. It is also hard to be affected by the progress.

Example 4: GanttProject[2] is a graphical Java program for editing Gantt charts. In the early part of 2005, updates which scored relatively high impCR value are the ones of refactoring about motion listener, mouse event, input file, resource, and so on. In other section, refactoring which arrange use relation or data structures also yields a high impCR value.

4.2.2 Application Result: JBidWatcher

As an individual application result, we show the case of jbidwatcher[7]. Jbidwatcher is Java-based auction monitoring and management software. In the experiment, it analyzes 303 revisions from 31st May 2000 to 27th Jan 2006. Figure 5 shows transitions of LOC and other metrics. We can grasp that these metrics are monotonically increasing with similar course.

Figure 6 shows transitions of impCR and number of components, and Table 2 shows a list of updates that yield high impCR, respectively. These results show that updates in early stage of the development marked high impCR due to implementations of fundamental and/or crosscutting features and data structures. In the case of jbidwatcher, implementation of browsing features, password login support, and dump facility are applicable to these updates.

On the other hand, the value of impCR gradually falls as the progress, and this means that existing use-relations are gradually stabilized. However, updates which include

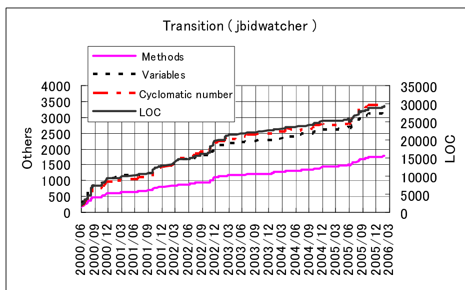

Figure 5. Transition of several metrics

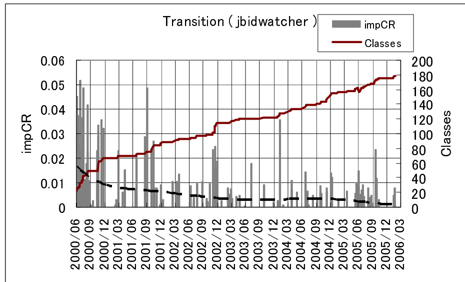

Figure 6. Transition of impCR and number of components

changes of class structure and data structures, such as readjustment, removal, extract, and deletion, yield high impCR value. Besides updates in Table 2, readjustment of generic interface, and refactoring for tables and search features also mark high impCR.

In the case of jbidwatcher, updates which mark high impCR often add some new components. However, updates which add a lot of new components don't always mark high impCR. Updates, which include only local changes and which don't break existing framework, don't mark high impCR. In the case of jbidwatcher, addition of autocompletion and support for help file were such updates. We think impCR is effective for extracting updates which have a large impact on the entire system.

5. discussion

We think that the proposed metric can contribute for process management. As the assumed hypothesis, component rank drastically changes as a response to the important update which contains a structural change. In postmortem like our experiment, the metric can capture important updates on development.

By comparing an actual impCR value with assumed impCR value, our metric can also give a signal of a serious problem. If we observe abnormal values with frequency, something bad may happen. For example, we consider the case that we observe updates whose impCR values are very high in near delivery of the system or a final stage of test phase. In such situation, we can easily imagine a situation in which the modified use relation is not tested, and the product includes serious bugs. In another situation, we consider the case that impCR of a mere function implementation doesn't converge even if at the latter term of the development. In this case, we can assume a situation that we must change various components in association with function implementation because independencies among components are very low. We may use impCR as an indicator for occasion of refactoring.

Table 2. High impCR Updates

	Date	impCR	Major Contents * Implementation
1	2000/6/23 (15th)	0.0516	Adjust mouse action Delete meaningless codes
2	2001/8/31 (70th)	0.0488	Restructure of UI Remove generic constants
3	2000/7/11 (20th)	0.0486	Handle a proxy server Add generic constants
4	2000/6/3 (3rd)	0.0454	* Browsing
5	2000/8/12 (34th)	0.0420	Extract logging classes * Visual display
6	2000/7/13 (22nd)	0.0400	Password login support * Bidding features
7	2000/6/6 (5th)	0.0372	* Splash screen
8	2000/7/8 (19th)	0.0367	* Menu bar
9	2000/11/4 (48th)	0.0359	Use Auction Manager * Multiple tabs
10	2004/1/4 (164th)	0.0352	Clean up platform -specific code

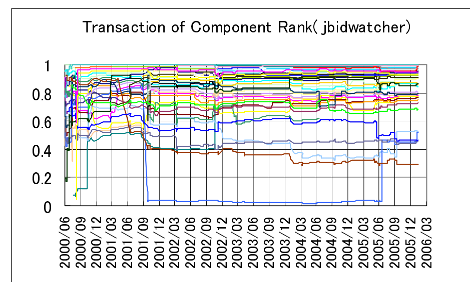

Figure 7. Transision of Component Rank

We think transitions of component rank of each component also have a lot of useful information. Figure 7 shows

transitions of component rank of highly ranked components in jbidwatcher. It's difficult to draw a significant conclusion from only the transition of each component; however, we can read out a dynamic and stabilized period (or point) about component rank throughout the whole graph. At the early stage of the development, component rank changes often and all curves intertwine with each other. However, rank's change stops at certain development points and components keep their rank for a while. We investigated a trigger for this phenomenon, and found that this is because of updates which include refactoring, rewriting and restructure activities. If components are correctly modified, developers don't retouch them for a while after such activities. We think we can use the transitions to confirm whether refactoring activities were actually effective.

As easy metrics for impact-measurement, you might consider that transitions of change of well-known metrics are also available. However, such transitions are not so significant if we could obtain a transition of LOC. This is because transitions of such metrics have a very similar tendency as the growth of LOC as shown in figure 5. ImpCR also has similar tendency as the amount of changed LOC, however, is better metric for detecting important updates which affect the whole software system because of the consideration of the change of use-relation.

As related works, there are several researches for software repositories to understand reasons of software changes[3], to identify how communication delay among developers have effects on software development[4], to detect potential software changes and incomplete changes[14], and so on. In [9], Johnson proposed an approach that records developer's work by installing sensors in her computer. The objective of this work is to find a relation between the internal characteristics (size and time, etc.) and the external characteristics (quality and reliability of products, etc.) rather than to measure update.

EPM[10] analyzes accumulated data on a configuration management system, a mailing-list system, and a bug-tracking system. The result is shown by graphical representation. The aim of EPM is sharing information about ongoing development project among developers and managers, by extracting effective metrics. We think EPM is effective tool for an objective grasp of progress and current status of the project. However, on the EPM, the analysis for source codes is only LOC and so insufficient. We think that EPM provides more significant information by integrating our method.

6. Conclusion

In this paper, we suggest an update-evaluation model based on component rank. In the model, we use the degree of change of each component rank as a quantified measure of impact about source code update. The application result to open source projects shows that our metric can capture important updates which include major internal changes, such as refactoring, changes to core component, and so forth. We think the proposed metric can contribute for process management by using it for postmortem and monitoring. As future works, we are planning a refinement of the proposed metric and further application to projects in diverse development styles.

Acknowledgments

This research project is supported by Pache Research Subsidy 2006 I-A-2 in Nanzan University.

References

[1] Galleon. `http://galleon.tv/`.
[2] Ganttproject. `http://ganttproject.sourceforge.net/`.
[3] 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.
[4] 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.
[5] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. “Component Rank: Relative Significance Rank for Software Component Search”. In *25th International Conference on Software Engineering (ICSE2003)*, pages 14–24, Portland, Oregon, 2003.
[6] 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.
[7] JBidWatcher. `http://www.jbidwatcher.com/`.
[8] JEdit. `http://www.jedit.org/`.
[9] 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.
[10] M. Ohira, R. Yokomori, M. Sakai, K. ichi Matsumoto, K. Inoue, and K. Torii. “Empirical Project Monitor: A Tool for Mining Multiple Project Data”. In *International workshop on Mining Software Repositories(MSR2004)*, pages 42–46, Edinburgh, Scotland, UK, 2004.
[11] PMD. `http://pmd.sourceforge.net/`.
[12] SOURCEFORGE.net. `http://sourceforge.net/`.
[13] SPARS-J demo. `http://demo.spars.info/`.
[14] 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.