I.J. Information Engineering and Electronic Business, 2015, 1, 49-58
Published Online January 2015 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijieeb.2015.01.07

A Content Assist based Approach for Providing Rationale of Method Change for Object Oriented Programming

Amit S. Ami
Institute of Information Technology, University of Dhaka, Dhaka, 1200, Bangladesh
Email: amit.seal@iit.du.ac.bd

Md. Shariful Islam
Institute of Information Technology, University of Dhaka, Dhaka, 1200, Bangladesh
Email: shariful@iit.du.ac.bd

Abstract—Software engineering requires modification of code during development and maintenance phase. During modification, a difficult task is to understand rationale of changed code. Present Integrated Development Environments (IDEs) attempt to help this by providing features integrated with different types of repositories. However, these features still consume developer's time as he has to switch from editor to another window for this purpose. Moreover, these features focus on elements available in present version of code, thus increasing the difficulty of finding rationale of an element removed or modified earlier. Leveraging different sources for providing information through code completion menus has been shown to be valuable, even when compared to standalone counterparts offering similar functionalities in literature. Literature also shows that it is one of the most used features for consuming information within IDE. Based on that, we prepare an Eclipse plug-in and a framework that allows providing reason of code change, at method granularity, across versions through a new code completion menu in IDE. These allow a software engineer to gain insight about rationale of removed or modified methods which are otherwise not available in present version of code. Professional software engineers participated in our empirical evaluation process and we observed that more than 80% participants considered this to be a useful approach for saving time and effort to understand rationale of method change. Later, based on their feedback, the plug-in and framework is modified to incorporate chronological factors. We perform quasi experimental evaluation with professional software engineers. It is found that time required to find rationale of method change is reduced to at least half compared to usual amount of time required for all the software engineers who participated in the quantitative evaluation.

Index Terms—Software engineering, Mining software repositories, experimental software engineering.

I. Introduction

Software engineering consists of several phases including software development and maintenance phase. Throughout these phases, modification of code takes place. There can be several reasons for code change, such as re-factoring, dead code removal, introducing design pattern, bug fix and new features. Code change can be categorized to different levels of abstraction, such as modifying, introducing or removing control logic, method in classes, renaming of local variables, and different types of re-factorings. Frequent changes were found to be made at the method signature level and control logic [1]. However, control logics work at fine granular level and generally do not have any identifier associated with it. Methods, however, are an integral part of object oriented programming. Moreover, fine level method signature changes require changing the method call at all sites, therefore having more impact than any other change kinds [2].

Even though there are many plug-ins and features to assist the software engineers in different ways, they still have to face different types of problems. At least 50% software developers consider that finding out reason of a code change is a difficult problem [3] [4]. It is because the general process of finding out reasons requires going through source code repositories and *issue* trackers.

"Issue" is a metaphor that represents bug, defect, ticket, feature, etc. [5]. However, this approach consumes developer's work hour, up to 50% of daily activity [6]. Going through commits generally takes a hit or miss approach. However, going through this is unavoidable in order to avoid collision between developers' activities. Advanced development works, such as troubleshooting unexpected code behaviors [7] and monitoring evolution of code are dependent on understanding code changes as well. Understanding rationale of a particular code segment change requires answering the following three questions:

- What part of code was changed?
- Why it was changed?
- Who changed it, in case further discussion is required

Software engineer's need to understand code change increases with time as the code base increases. If he does not know when a code segment he is interested in was changed, the task gets complicated. Even though it is possible to get some idea about the changes that took place through *commit* histories, the procedure of finding related commit is time consuming. Moreover, this requires the developer to switch focus from editor view of IDE, thus distorting attention. This whole procedure is depicted in Fig 1. Sometimes, a developer may look for a code element that was removed in a previous version. To the best of our knowledge, existing approaches do not directly provide rationale of changed or removed elements and requires detailed, time consuming investigation utilizing version control tools.

There are several research works found in literature which focused on establishing connection between software artifacts [3], [8], [9], [10]. These aimed to assist software engineer during development and maintenance phase. Assistance is generally about finding that made the change, when it was changed and what changed. Most of these approaches provide assistance outside editor view in IDE. Approaches that provide assistance in editor focus on current version of code. However, few works in literature aim to assist a developer understand rationale of code change in editor.

We aim to provide this information to developer in a convenient way through the combination of a framework and an IDE plug-in. To find the most attractive way of providing information to developer, we conduct a survey to find the most useful feature of IDE and how frequently it was used [11]. Unsurprisingly, code completion, a feature based on content assist came at top with 68% votes as the most used feature of IDE. Version control and debug received 23% and 5% vote respectively. 76% participants voted that they use it very frequently during development, as shown in Fig 2. This is consistent with the survey by Murphy et al. [12] who observed that code completion menu is one of the top six features used in Eclipse IDE for consuming code specific information. Features voted above code completion in his observation were basic editing functionality whereas our survey focused on features related to consuming information specially provided through IDEs only.

We propose an approach of providing reason of code change related information in the IDE through a separate code completion menu. The framework allows extracting information from source code and issue repositories. The prototype plug-in provides information related to code change to the developer on the fly. In case a relevant issue id is not found, the plug-in provides what changes took place and when it took place.

To determine whether our approach is useful for developers, we implemented the framework and prepared a plug-in for Eclipse IDE for Java [13]. The usefulness of this approach is evaluated by professional software engineers experienced in object oriented programming using Eclipse IDE following empirical software engineering guidelines. More than 80% software engineers were found to be satisfied with its utility [14] in qualitative evaluation and expressed their opinions. These are described in our previous work in [15]. Based on their feedback, the framework is modified and the plug-in is improved to reduce information overflow, which are described in this work. Additionally, our quantitative evaluation of this approach is described in this work as well.

We report the following contributions through this paper:

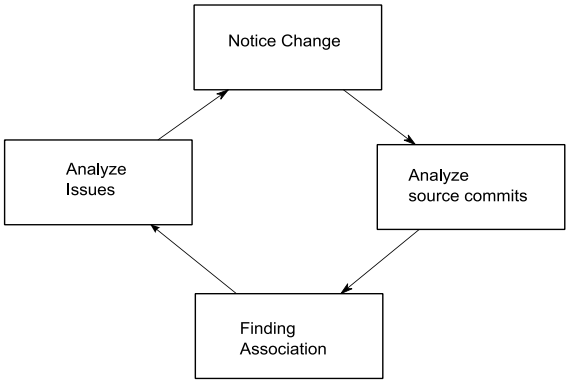

Fig 1. General Process of finding rationale of code change

- A unique approach to help developer understand code change from within IDE even when an item of interest is not present in the current version of code
- An extensible framework that provides a guideline to collect information across repositories and provide code change related information through the IDE

Both of which helps answer the what, why and who changed a code segment as described earlier, thus providing rationale of code change to software engineer. We have discussed works which we found relevant in Section II. In Section III and IV, we described the framework requirements as well as the details about the Framework. In Section V and VI, descriptions of evaluation in qualitative and quantitative approach are described respectively. Finally, our concluding remarks and scope of future works are discussed in Section VII.

II. Related Works

Several works have been done to make understanding code change easier for developers. Giving this type of information requires mining software repositories. We have considered research works that mine software repositories to assist developers understand code change or find artifacts related to code change as related work.

Lee et al. [9] first introduced the idea about adding temporal dimension in code completion. The proposed idea was that code changes can be shown through code completion. They also allowed users to go to the previous versions of code through temporal code navigation. However, it focused on the code changes only, rather than the reasons of code change.

Venolia [8] created Bridge framework, that aims to establish connections between software artifacts by means of textual allusions and preparing a graph based system. However, it did not focus on providing reasons of code change to developer.

Although not focused on providing reason of code change to developer, Codetrail by Goldman et al. [16] established connection between IDE and artifacts such as documentation, bug fixes and error descriptions available in web.

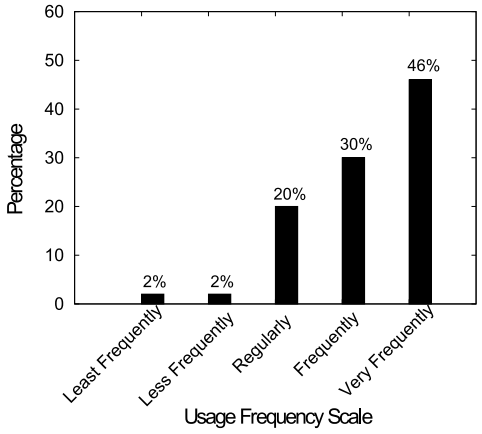

Fig 2. Frequency of using auto complete feature in IDE

Begel et al. [3] created CodeBook framework that mined software repositories relying on the concept of Bridge Framework.

They aimed to solve several problems of software engineers, which included helping, understand why a recent change was made. However, it focused on present version of code. Hipikat is a recommender like system by Čubranić et al. [17] [18]. It mined software artifacts across different repositories to give access to project memory and recommends artifacts. Their approach involved querying the Hipikat system from the IDE and then explore through artifacts.

Rastkar et al. [10] proposed that reason of code change can be found by summarizing multiple documents which are related to the change. They proposed that machine learning can be used to identify appropriate sentences in documents.

Deep Intellisense by Holmes et al. [6] utilized Bridge framework. They provided the "how and how" of code change to developers. Their approach consisted of three views about code items that offered history, people and event related information in the IDE.

Voinea et al. [19] worked on representing code history visually through lines as well as the contribution by individuals for CVS version control management system. It required the developer to move away from the editor and focus on separate software to find out what happened over the versions.

Local History [20], a facility provided by IntelliJ Idea allows navigating through different versions of personal code changes from within the IDE, while pointing out the additions, removals, and modifications. It also helps by putting labels on each version. However, it is not for version controlled source code repositories and shifts attention of developer from the active editor part of IDE.

Yao et al. [21] worked with CVS repositories used by open source community at that time. A system was provided which allowed searching through different source code versions extracted from CVS comments. They mainly focused on providing service about this information to developers.

LSDiff, which represented logical structural difference, was proposed by Kim et al. [7] This allowed inferring system changes by abstracting a program as code elements consisting of methods, fields and structural dependencies between these.

Version Editor (VE) by Atkins et al. [22], introduced the concept of giving information related to each line in source file which helped developer know when that line was created, who created as well as why it was created.

To evaluate our approach of providing reason of code change through IDE, we provide information related to code change. To achieve this, a customized database is required which will contain the required information to be displayed in the IDE through the prototype plug-in. Collecting these data and providing these in the form of useful information requires the preparation of a framework. Collecting these data and providing these in the form of useful information requires the preparation of a framework. We have prepared a set of requirements that the framework should fulfil and discussed this in Section III.

III. Framework Requirements

A framework that can seamlessly provide issue based reasoning of code change has to be connected with source code version control and issue repository (or repositories). A series of considerations are required in order to make it an extensible framework.

The framework mines software repositories to establish relation between code changes, code repository as well as the issue repository. The mining process should be fast and light enough to be setup and executed quickly. This requires mining to take place only from useful resources or entities. This also requires establishing relation with the related entities in other repositories. It should be flexible to support different types of repositories. It should be extensible, so that anyone can extend the framework and change the mining process for gathering more information. Moreover, the mining process should take place only when new changes are available in code repository.

How the framework will expose itself is also dependent on whom it is being exposed to. For example, IDE users might seek reasons of code change while understanding code change of closed source libraries. The framework should provide a public interface in such cases based on configuration. This would allow the library users to know reason of code change while sensitive information is encapsulated.

The means of providing information should be easy, fast and simple to avoid making the user go through extra

steps. Information provided to the user should be adequate enough to give him an understanding of code change that took place. Moreover, it should be provided in a compact format. It should satisfy user's need or at least give him a starting point to search for further details.

Lastly, the information should be made available to user in real time. If a user has to wait for information to pop-up because data is not ready on the fly, the work flow of the user will be hampered. This should not happen, because we aim to assist during development process by reducing the time required to find reasons of code change.

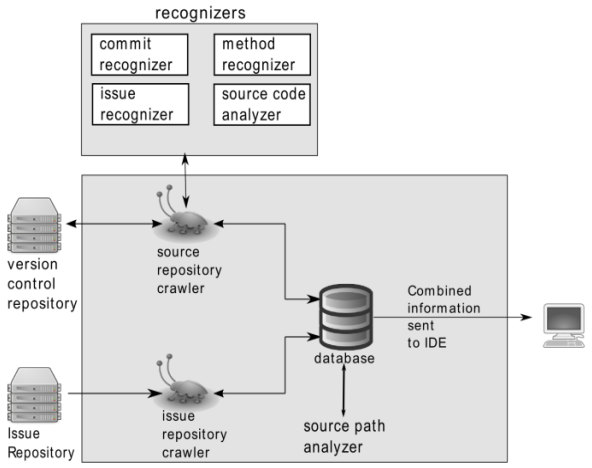

Fig 3. Framework for IBRICC

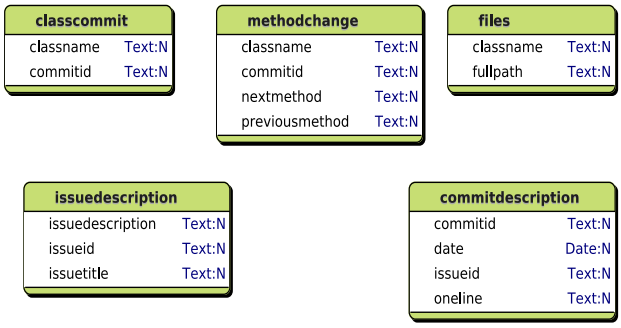

Fig 4. Framework schema for IBRICC

Based on the discussion above, we prepare a customized framework which allows us to mine repositories and populate the database with information in our desired format. The framework we have prepared, Issue Based Reasoning in Code Completion (IBRICC), is detailed in the following Section IV.

IV. IBRICC FRAMEWORK

IBRICC consists of several components, including a set of crawlers and plug-in. The components are utilized to populate the framework's own database with information related to changes made to source codes. The data stored in database can be made available by providing a service. We have used the service to provide a local standalone database; using SQLite3 [23] engine, in order to provide information to user in real time. An overall architecture of the framework is provided in Fig3.

A. *Source Code Repository Crawler*

The source code repository crawler crawls the repository in several stages. This can be done by analyzing the source code repository database, or by utilizing source code repository API (Application Programming Interface). At first stage, it goes through the current version of code repository in order to find current source code files available. Additionally, it scans through these files in order to find the classes in those source files. This information is stored in database. At the second stage, it goes through the commit histories available of each file in order to find commits associated with each class. This allows relating the commit ids with the files, including composite changes, i.e. same commit ID relating to multiple source files. For example, the following code segment gives output to the commits previously made for the file name provided to it.

```
git log --oneline Lexer.java
be0d6b3 clean up equals/hashcode for Interval
(Coverity)
a7a2050 rm dead code (Coverity)
```

These data, establishing relation between commits and classes are stored in database. In order to find associated issues, a set containing all commit IDs related to all source files is analyzed along with commit descriptions associated with it. Commit descriptions are analyzed utilizing rule based approach in order to identify issue related terms, such as Bug #12345, Fixes #1234, Issue #12345 etc. The found issue IDs are tagged with the related commit IDs. For example, the given code segment can be used to find description of a particular commit.

```
git show -s ---format=%B 7049972
escape \r \n \t in lexical error messages.
Fixes antlr/antlr4#75
```

Finally, each commit ID related to available source files are utilized in order to find the different versions of the source files. Text based differential tools, such as *git show*, are used to identify textual changes introduced per commit per source file, which is based on string matching [24].

```
git show 7049972 Lexer.java
- public String getCharErrorDisplay(int c)
+ public String getErrorDisplay(String s)
```

As shown, differential tools can be configured to identify only the changed portion of text through versions. This can be further analyzed to classify changes introduced. As shown in previous code snippet, we have used *git show* to identify a method declaration modified in a particular commit. Change details, including previous and committed version of code segments, related commit id and related classes are stored in database.

B. *Issue Repository Crawler*

As mentioned in Section IV - A, the issues IDs are identified in advance. Based on the project setup, these issue IDs are classified. The task of Issue Repository crawler is to gather data about those issue IDs. Gathering data can take place by directly communicating the repository database or by utilizing the issue repository API. For example, the following code snippet in Python utilizes PyGithub library [25] in order to gain issue details from `https://github.com/issues` by specifying an issue ID. In the following code snippet, a particular GitHub [26] repository is defined in constants.git_repo.

```
G_INSTANCE = github.Github()
REPO = G_INSTANCE.get_repo(constants.git_repo)
github_issue = REPO.get_issue (issue_id)
```

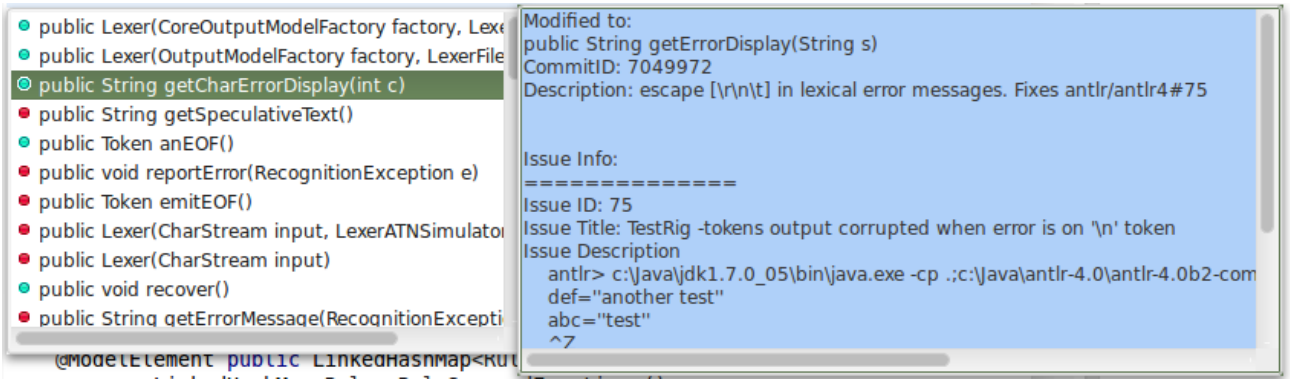

Fig 5. Prototype plug-in in Eclipse displaying rationale of method change

Table 1: Statistics Found through Framework Implementation

Repository	Modified Methods	Removed Methods	Found Issues	SQLite DB Size
Antlr4	1132	2419	53	942KB
K-9	661	7049	29	1.3MB
JUnit	2796	281	117	4.3MB

C. *Database*

The database is used to store information related to changes, classes found from latest version of available source files as well as the commit descriptions. Five schemas, as shown in Fig.4, are used to store the required information.

classcommit schema holds information related to the commits available for a class by relating unique commit IDs with classes.

commitdescription schema contains commit related details, i.e. the unique commit ID, description related to it. If any issue ID is found from commit description, its descriptions as well as title are stored as well.

methodchange schema contains data methods changed, i.e. removed or modified through commits and tags these along with the *classnames*.

issuedescription schema holds information related to issues found by analyzing commit descriptions. For each issue id found, relevant issue description and issue title is collected and stored.

files This schema is not directly related to the framework in the sense that it is not required providing rationale of code change to software engineer. However, this is necessary for listing the necessary source files and their paths for executing the mining process.

With a combination of these four schemas, it is possible to point out every method change in source files throughout different versions and provide relevant information to user on the fly.

D. *IDE Plug-in*

To evaluate our approach, we have prepared a plug-in for Eclipse IDE. Its purpose is to provide information related to rationale of code change through code completion, covering method changes. This should not be directly mixed with normal code completion. Otherwise, a user may have to scroll through a large number of code completion proposals and code change related proposals. The information is provided through a separate code completion menu. As a result, a user can see reason of code change only when he intends to. Next, user should be made aware that method changes are of two types, removal and modification. Therefore, the code completion items should be differentiated visually. One indicator is used to indicate the methods which are changed in a commit and the other one is to indicate methods which are removed through a commit, as shown in Fi 5. Therefore, the information provided through the plug-in are as follows:

- List of methods removed or modified in the class
- If a method is modified, the modified method which is introduced in its place
- Commit ID for each method change item
- Commit description related to commit ID
- Issue id, if found from the commit description
- Issue title, if issue ID was found item Issue description, if description of issue ID is available
- Issue description, if description of issue ID is available

User is able to gain rationale of method change in several approaches. These approaches allow the user to

gain information about their method of interest without risking information overload.

- Generic Approach

In this approach, user simply invokes the plug-in within the IDE and the plug-in shows list of available modified methods to user in the IDE.

- Keyword based Filtering Approach

In this approach; user specifies a keyword when invoking the plug-in, for example, a portion of the modified method he is searching rationale for. The plug-in filters the information automatically and displays information within the IDE accordingly.

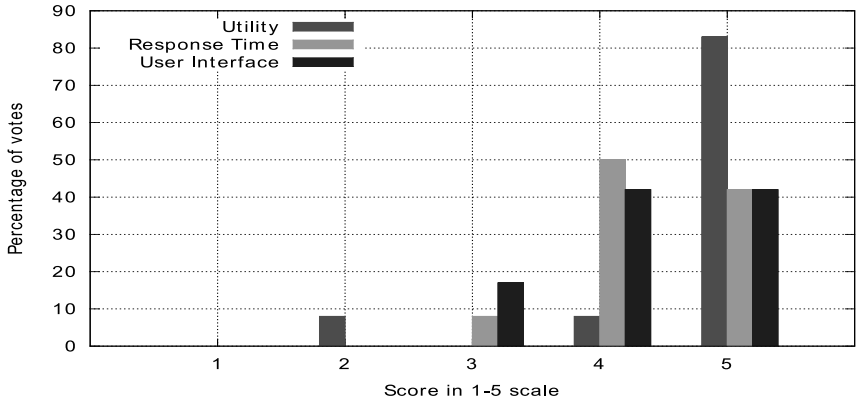

Fig 6. Plug-in Evaluation through Survey

- Chronological Filtering based Approach

Since it is possible that user is interested in a method signature that is no longer present in the current version of code, as it was renamed or modified in any other approach - user can filter the data of generic approach using a time range. The plug-in automatically filters the information of the generic approach and provides information accordingly on the fly.

- Keyword and Chronological Filtering based Approach

This approach is a combination of the keyword based filtering approach and chronological filtering based approach.

V. Lab Evaluation

Based on user study in a controlled, off-line environment, the evaluation focused on experience by software engineers while utilizing the plug-in in IDE to find out rationale of method change. Several open source repositories from GitHub were utilized. The users were not familiar with changes that took place previously. Evaluation process involved utilizing the plug-in, asking open ended questions, survey and discussion between authors, participants.

A. Study Setup

Three GitHub Java repositories were utilized for evaluation. The repositories we have chosen for analysis were:

1. Antlr4, Another Tool for Language Recognition version 4 [27]
2. Storm, a real time distributed computing system [28]
3. k-9, an advanced mail client for Android [29]

These open source repositories are highly active in GitHub open source repository, where open source programmers are contributing from different parts of the world for over two years. Eclipse IDE for Java Developers was prepared with the pulled in git repositories. The prototype plug-in was installed in Eclipse IDE. Each project contained one SQLite database prepared in advance using the IBRICC framework.

A. Participants and Data

The collected repositories consists a total of 844 source files, 12788 commits. Details about information gathered through the Framework of individual repositories are provided in Table 1.

We recruited the participants from software engineering firms as well as graduate software engineers from IIT, University of Dhaka through mailing lists and personal communication. Of the twelve voluntary participants, 50% were professional software engineers and 50% were exceptional students accomplished in software engineering competitions. The later group had at least six months of professional software engineering experience in software firms. All of them were experienced with Java programming language, Eclipse IDE for Java Developers, version control and object oriented programming concepts. The mix of professional and graduate students allowed us to evaluate the plugin from both experienced and novice programmer's point of view.

B. Result

Compared to number of total commits, the number of commits that can be used to identify changes in files available in current version is comparatively lower. This is observed for several reasons.

- Files are removed throughout versions. Because source files from the latest version only was considered, some files which contained changes through commits were not crawled.
- Commits are used to make changes in non-source code files, such as readme, log and version. This also increased the number of commits.
- Developers often commit unrelated or loosely related code changes in a single transaction, thus reducing the number of commits correctly associated with issues [30].

Again, it is observed that there are few issues compared to number of commits that covers all source code versions. This is due to the reason that issues are crawled using regular expression based filters searching for `#' in commit description. Additionally, GitHub repository contributors often include enough information through commit description due to the lack of integration of a separate issue repository, thus reducing its number.

Table 2: Quantitative Evaluation of IBRICC

Developer	Avg. Time – 1	Avg. Time - 2	Avg. Commits
1	45	5	3
2	204	15	8
3	125	20	5
4	224	15	8
5	87	20	6

Before evaluation, the participants were introduced to the plug-in and its usage in a thirty minute interactive session. They asked questions about the framework and whether it could be extended to support other types of repositories. In all their plug-in invocation, at least what, and who related information was provided. In some cases, information extracted from issue repository was not available, however - that was due to lack of textual allusion in commit description. Some common interview questions from software engineers are:

1. *How is the plug-in displaying this information?*

Software engineers were curious to know the techniques used behind the tool, whether it was collecting and providing information on the fly or were showing information from a predetermined storage.

2. Can I reduce or order the number of items in code completion?

The prototype plug-in considered all the changes made from the start of project. Users may or may not be interested in older changes. Therefore, they asked whether it was possible to filter or order code completion items.

3. Is this applicable for distributed VCS only?

Distributed VCS features local version of code repository, thus helping to do faster commits and merging. Software engineers were curious about the performance and configurations necessary for applying the framework in a centralized version control system scenario.

During the over forty minute evaluation, software engineers were interviewed after spending their time going through the projects from GitHub repositories, commit histories and utilizing the IDE plug-in. Participants randomly choose source code files and invoked the special code completion feature to gain insight about changed method definitions. They were asked to give feedback based on Utility, Response Time and User Interface in a scale of 1 to 5, where greater value means better. The survey results are depicted in Fig 6.

VI. Quasi Experimental Evaluation

To further evaluate the approach of providing rationale of code change in the IDE, we conduct quasi experiment with the help of six professional software engineers following the Rapid Application Development model of software engineering. The aim of the experiment is to initiate an environment as close to work environment as possible and measuring the performance benefits by utilizing the framework and the plug-in in the software engineering process.

A. Participants

The six participants are professional software engineers with graduation degrees from software engineering institute and working in various software engineering firms with a minimum of one year of experience. All of them are interviewed to ensure that they are familiar with and follows or utilizes proper object oriented concepts, distributed version control system and collaborative software engineering. Through interview, we ensure that the participants are familiar with:

- Eclipse IDE
- Git Version Control System
- Issue repository system in GitHub
- Git version control integration facilities in Eclipse
- Object Oriented Concepts and programming facilities of Java Programming Language

B. Lab Setup

The experiment followed the within subjects model to avoid biasness in performance comparison of software engineers. The experiment takes place for a total of 126 work hours. The participants are asked to create an inventory management system and are given specific requirements about the management system. They are also given instructions to follow de facto standards of commit rules, which are:

- Making commits for small, individual changes in source files
- Before making changes in another person's code,

raising an issue in the issue repository when necessary
- Referring to issue ids in case a commit fixes it
- Provide adequate information in commit description and issue description, if an issue is made

Each participant uses their personal computers and installed Eclipse, Git version control system in their preferred operating systems, i.e. Windows or Ubuntu. The participants setup a free, open source GitHub repository under an organization for the project to collaborate between themselves. Each participant decided to work on distinct components of the system after discussing between themselves.

To introduce code changes, we ask participants to review each other's components at code level. Based on the system design documents prepared beforehand by them, we asked them to make any changes necessary for ensuring that proper object oriented concepts are being followed. After this, we identified the changes made in the signature level of code and separated these to two groups. We asked software engineers to check their own components after the changes are made. The software engineers are not informed about the changes made and is asked to identify the changes. After identification, we ask them to analyze and understand half of the changes they considered confusing by the usual procedure of utilizing version control tools offered by the Eclipse IDE and Git tools. The whole procedure of understanding each change is timed for each software engineer. For the rest of the changes, we ask them to understand changes by utilizing the plug-in after the dataset is extracted by mining software repositories using the framework. They are given a brief introduction about the plug-in and a practical demonstration of its usage in a separate project to avoid learning curve related time consumption.

C. Measurements

Because of the limited scope of the quasi experimental software project, as well as strict following of standard software engineering guidelines, precision and recall rate of the framework become very high. Instead, to evaluate the proposed approach, performance of software engineers in terms of time consumption while understanding code change is measured. During measurement, out of the six software engineers, result of one software engineer is discarded as there was no significant code change that raised confusion. For the other five software engineers, the average time for understanding code change in usual approach (Avg. Time - 1), the average time for understanding code change utilizing the framework (Avg. Time - 2) and the average number of commits the developer had to analyze (Avg. Commits) are provided in Table 2.

D. Discussion

As mentioned previously, the quasi experiment was strictly done to compare performance of software engineers in terms of time while finding rationale of code change within the IDE between the usual procedure and while being facilitated by the framework. Moreover, the duration of the project was rather short, around 126 work hours, which resulted in a rather short amount of development. The mining approach is based on static code analysis and textual analysis, which is shown to be less effective by Herzig et al [30] for older software projects which does not follow the atomic commit convention. However, this evaluation gives us the notion that with the utilization of framework, the time consumed to browse through different commits just to find a particular code change and its rationale can be drastically reduced.

VII. Conclusion And Future Work

In this paper we have introduced the concept of providing reasons of code change through code completion along with code changes across versions. Software engineers require knowledge about code change - what, when and why the code changed. Our proposed approach aims to answer these questions right within the editor view of IDE. We have conducted a survey and found that code completion is the appropriate approach to provide this type of information. We have implemented the framework and a plug-in to gain feedback from software engineers. From evaluation, it was found to be very useful in terms of utility and response time. The evaluation of framework included analyzing source code repositories to see whether it was possible to establish relations between different types of repositories and display a summarized result. It was found that simply utilizing regular expression to find reasons of code change from commit descriptions may not return enough results. However, the main contribution of our paper was to provide the "what, when and why" related information of method change to software engineer within the IDE. Through evaluation by experienced software engineers, we found that our approach is useful from a software engineer's perspective and can save valuable time. One of the feedbacks that we consider significant is that it should allow software engineers to navigate to the details of changes from the IDE.

In conclusion, providing reasons of code change through code completion menu along with removed or modified elements in code is an unexplored field we need to turn our attention to. When properly utilized, it can save time during software development and maintenance phase by helping software engineers understand code change from within the IDE without interrupting their work flow.

Acknowledgment

This paper is based upon work supported by the fellowship from ICT Division, Ministry of Posts, Telecommunications and Information Technology, Bangladesh.

References

[1] Steve Counsell, Youssef Hassoun, Roger Johnson, Keith Mannock, and Emilia Mendes, "Trends in Java Code Changes: The Key to Identification of Refactorings?," in *Proceedings of the 2Nd International Conference on Principles and Practice of Programming in Java*, New York, NY, USA, 2003, pp. 45-48. http://dl.acm.org/citation.cfm?id=957289.957305.

[2] Sunghun Kim, E. James Whitehead, and Jennifer Bevan, "Analysis of Signature Change Patterns," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1-5, may 2005. http://doi.acm.org/10.1145/1082983.1083154.

[3] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann, "Codebook: discovering and exploiting relationships in software repositories," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, 2010, pp. 125-134. http://doi.acm.org/10.1145/1806799.1806821.

[4] Thomas D. LaToza, Gina Venolia, and Robert DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 492-501. http://doi.acm.org/10.1145/1134285.1134355.

[5] Reid Holmes Olga Baysal and Michael W. Godfrey, "Situational Awareness: Personalizing Issue Tracking Systems," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[6] Reid Holmes and Andrew Begel, "Deep intellisense: a tool for rehydrating evaporated information," in *Proceedings of the 2008 international working conference on Mining software repositories*, New York, NY, USA, 2008, pp. 23-26. http://doi.acm.org/10.1145/1370750.1370755.

[7] Miryung Kim and David Notkin, "Discovering and representing systematic code changes," in *Proceedings of the 31st International Conference on Software Engineering*, Washington, DC, USA, 2009, pp. 309-319. http://dx.doi.org/10.1109/ICSE.2009.5070531.

[8] Gina Venolia, "Textual Allusions to Artifacts in Software-Related Repositories," in *Proceedings of the 2006 international workshop on Mining software repositories*, 2006, pp. 151-154.. http://doi.acm.org/10.1145/1137983.1138018.

[9] Yun Young Lee, Sam Harwell, Sarfraz Khurshid, and Darko Marinov, "Temporal code completion and navigation," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 1181-1184. http://dl.acm.org/citation.cfm?id=2486788.2486956.

[10] Survey on IDE (Responses), http://goo.gl/78Ye5G.

[11] Sarah Rastkar and Gail C. Murphy, "Why did this code change?," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 1193-1196. http://dl.acm.org/citation.cfm?id=2486788.2486959.

[12] Gail C. Murphy, Mik Kersten, and Leah Findlater, "How Are Java Software Developers Using the Eclipse IDE?," *IEEE Softw.*, vol. 23, no. 4, pp. 76-83, #jul# 2006. http://dx.doi.org/10.1109/MS.2006.105.

[13] Eclipse IDE for Java Developers, http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/keplerr.

[14] Survey on Eclipse Plugin of Issue based Reason of Code Change (Responses), http://goo.gl/9Dq8yr.

[15] Amit Seal Ami and Md. Shafirul Islam, "An Efficient Approach for Providing Rationale of Method Change for Object Oriented Programming," in *International Conference on Informatics, Electronics Vision (ICIEV)*, Dhaka, 2014, pp. 1-6.

[16] Max Goldman and Robert C. Miller, "Codetrail: Connecting source code and web resources," J. Vis. Lang. Comput., vol. 20, no. 4, pp. 223-235, #aug# 2009. http://dx.doi.org/10.1016/j.jvlc.2009.04.003.

[17] Davor Čubranić and Gail C. Murphy, "Hipikat: recommending pertinent software development artifacts," in Proceedings of the 25th International Conference on Software Engineering, Washington, DC, USA, 2003, pp. 408-418. http://dl.acm.org/citation.cfm?id=776816.776866.

[18] Davor Cubranic, Gail C. Murphy, Janice Singer, and Kellogg S. Booth, "Hipikat: A Project Memory for Software Development," IEEE Trans. Softw. Eng., vol. 31, no. 6, pp. 446-465, 2005. http://dx.doi.org/10.1109/TSE.2005.71.

[19] Lucian Voinea, Alex Telea, and Jarke J. van Wijk, "CVSscan: visualization of code evolution," in Proceedings of the 2005 ACM symposium on Software visualization, 2005, pp. 47-56. http://doi.acm.org/10.1145/1056018.1056025.

[20] IntelliJ IDEA : Local History - protect your code from any accidental losses or modifications, even if made by other outside applications.

[21] Andrew Y. Yao, "CVSSearch: Searching through Source Code using CVS Comments," in Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01), Washington, DC, USA, 2001, pp. 364--. http://dx.doi.org/10.1109/ICSM.2001.972749.

[22] David L. Atkins, "Version sensitive editing: Change history as a programming tool," in ECOOP 98, SCM-8, LNCS 1439, 1998, pp. 146-157.

[23] SQLite Home Page, http://www.sqlite.org/.

[24] Miryung Kim and David Notkin, "Program element matching for multi-version program analyses," in Proceedings of the 2006 international workshop on Mining software repositories, New York, NY, USA, 2006, pp. 58-64. http://doi.acm.org/10.1145/1137983.1137999.

[25] PyGitHub 1.18.0: Python Package Index, https://pypi.python.org/pypi/PyGithub/1.18.0.

[26] GitHub, https://github.com/.

[27] antlr/antlr4, http://github.com/antlr/antlr4.

[28] nathanmarz/storm, http://github.com/nathanmarz/storm.

[29] k9mail/k-9, http://github.com/k9mail/k-9.

[30] Kim Herzig and Andreas Zeller, "The Impact of Tangled Code Changes," in Proceedings of the 10th Working Conference on Mining Software Repositories, Piscataway, NJ, USA, 2013, pp. 121-130. http://dl.acm.org/citation.cfm?id=2487085.2487113.

Authors' Profiles

Amit S. Ami received his B.Sc. in Information Technology (software engineering) from the Institute of Information Technology, University of Dhaka, Bangladesh in year 2012. He received his M.Sc. in Software Engineering from the same university in year 2014.

His research interests include experimental software engineering, mobile application engineering and testing, and mining software repositories. He has also worked on wireless mesh networks, wireless routing protocols, cloud computing and human

computer interaction. He has worked in software industries for a year, was a Microsoft Student Partner and ACM Student Member. He also worked as an organizer with Google Developer Group, Bangladesh and Mozilla Bangladesh. He joined as a lecturer at Institute of Information Technology, University of Dhaka in year 2014 and is currently working there.

Md. Shariful Islam received his B.Sc. and M.Sc. degree in Computer Science from the University of Dhaka, Bangladesh, in the year 2000 and 2002, respectively. He completed his M.S degree in Information Technology from the Royal Institute of Technology (KTH), Sweden, in 2005. He obtained his Ph.D. degree in Computer Engineering from Kyung Hee University, South Korea in February, 2011.

He is now working as an Associate Professor in the Institute of Information Technology (IIT), University of Dhaka, Bangladesh. His current research interests include the design of routing protocols, metrics and MAC protocols for wireless mesh networks. He also worked on security issues related to Wireless AdHoc and Mesh Networks. He has published a good number of research papers in international conferences and journals.

Mr. Islam is a member of IEEE and KICS.

Manuscript received July 13, 2014; revised November 11, 2014; accepted October 25, 2014.

How to cite this paper: Amit S. Ami, Shariful Islam,"A Content Assist based Approach for Providing Rationale of Method Change for Object Oriented Programming", IJIEEB, vol.7, no.1, pp.49-58, 2015. DOI: 10.5815/ijieeb.2015.01.07

