Who should `catch (this Exception) {?}`

Walaa ElDin M. Moustafa
Department of Computer Science
University of Maryland
College Park, Maryland, USA
walaa@cs.umd.edu

ABSTRACT

In this paper, we propose a software tool that can be used to speed up the software development lifecycle by automating the process of assigning new bugs to developers. Open bug repositories are now very common, and they are provided to the software users as a means to report any bugs they find while using the software. Many of the bugs reported complain about exceptions that are thrown by the program during normal operation of the software. Users report these bugs to the development team using bug reporting tools, (e.g. Bugzilla), and as a result of that, a person called *Bug Triager* tries to link between this bug and a developer who can fix it. Sometimes, the source of the bug is unclear, and as a result of that the developer can be *reassigned* several times. In this project, we propose an approach to automate this process by recommending developers who can fix a bug that complains from an exception. Moreover, we provide a ranking of developers according to their expected relationship with the bug so that more than one developer can work on the bug if needed, or if the first recommendation is unavailable. The approach is based on network analysis, and uses link prediction to rank developers with respect to a bug report. Moreover, in this report, in addition to using link prediction, we show several applications to using network analysis in the context of software analysis, like importance ranking and visualization. The Eclipse open source project was taken as a case study, and its CVS and Bugzilla repositories were the source of the datasets used to evaluate our approach. Experimental evaluation of the link prediction approach for recommending developers reveals its effectiveness and shows that its recommendations were similar to the actual bug assignments to a far extent.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management – *Life cycle, Productivity, Programming Teams.*

I.2.6 [**Artificial Intelligence**]: learning – *Concept learning.*

General Terms

Management, Measurement, Human Factors.

Keywords

Version Control, Bug Repositories, Network Analysis, Link Prediction.

1. INTRODUCTION

Software projects are consistently expanding in size, number of developers, and number of users dealing with them. Also, software development is spreading over more geographically distributed environments, where developers do not necessarily have the chance to meet. As a result of that, most state-of-the-art software projects maintain software repositories, or version control systems that keep track of all work and all changes done to project files, and allow several developers, potentially widely separated in space and/or time, to collaborate. On the other hand, as software projects become larger, software quality becomes a central concern. Therefore, the software development process involves different procedures for quality assurance, including static analysis, in-house software testing, in-field software testing, and bug tracking systems. Bug tracking systems are particularly interesting because they allow the software users to submit any negative experience with the software that looks like a bug, and notify the developers about it so that the bug can be fixed. In this paper, we are interested in analyzing information in version control systems, and information in bug tracking systems. We believe that these are very valuable information sources that not only record the state of the software, but also can reveal very useful information, if well handled and analyzed. In this paper, we are proposing a software tool that can be used to speed up the software development lifecycle by automatically assigning new bugs that complain about exceptions to the appropriate developers. This tool extracts CVS information regarding source code change history, and applies network analysis techniques on the extracted information, to infer rankings of all the developers with respect to their relationship with different methods in the source code. Link prediction techniques, which are a set of well established methods in network analysis, are used to obtain these rankings. For finding the ranking on the exception level, rather than the method level, method-level rankings are leveraged to compute rankings for developers with respect to all the methods that are reported in the exception. Moreover, in this paper, we also show several applications to network analysis in the context of software project analysis rather than developer recommendation. These applications include techniques for computing importance ranking, and visualization of the information extracted from CVS. We show that network analysis approaches are effective in understanding and mining new information about a software project. The rest of this paper is organized as follows. In section 2, we discuss the relevant background regarding version control systems and bug reporting tools, and focus on two popular tools, CVS and Bugzilla. In section 3, we discuss how to extract useful information form CVS, and show a way to get more semantic-oriented information about the source code changes, rather than pure text-based difference information that are supported directly by CVS. In section 4, we discuss how to employ the information extracted from CVS to construct a network that represents the relationship between the source code and the developers. In section 5, we show some applications for the network representation including importance

ranking, visualization, and discovering collaborations. In section 6, we discuss the different methods of link prediction in dynamic network analysis, and show how we employed link prediction to rank developers with respect to methods. In section 7, we show how to utilize method level rankings to create exception level rankings. In section 8, we show experimental evaluation of our approach. In section 9, we discuss related work. In section 10, we conclude by summarizing our work, and discussing possible future directions.

2. CVS and Bugzillas

In this section we will give brief and relevant background about version control systems, and bug reporting tools, focusing on CVS [1] and Bugzilla [2] as case studies.

2.1 Version Control Systems

Version Control Systems are commonly used in software development to manage ongoing development of relevant project files as application source code, documentation files and other information that may be worked on by a team of people. Most recent versions of the project files are stored in a central repository, and developers can check out copies of these files into their local workspaces, so that they can work on them, make changes, and apply these changes (commit) to the versions in the repository. Changes to these documents are usually identified by incrementing an associated number, termed the "revision number", and associated historically with the person making the change. A simple form of revision control, for example, has the initial version of a source code file assigned the revision number "1". When the first change is made, the revision number is incremented to "2" and so on [3]. Version control systems not only provide complete access to information about all the changes that took place in the project, but it also allows for full control over these changes, and allow developers to revert back a change, and go backward to an older revision if needed. Version control systems store different revisions in a compressed way, where only the differences from the previous revision are stored for a given revision. In addition to the information stored about files, their revisions, and the exact changes in each revision, version control systems store information about the developer that made a particular change, and the timestamp of that change. A very commonly used software tool for version control is Concurrent Versioning System, or CVS. CVS supports all of the features we have just discussed. For example, the command CVS log, shows all the files that are currently in the source tree, along with their all revision numbers since the file was initially created and the developers that created each revision, along with the timestamp of the change. The command CVS diff takes two revision numbers as a parameter, and returns text-based differences between the two revisions. This is usually not helpful in analyzing differences on the level of source code, because when analyzing source-code-level differences, we are more interested in finding semantic differences. For example, we may be interested in finding which methods have been changed in a class, what data members have been added, or which classes had their access qualifiers changed, while we might not be interested in information about adding or deleting comments, or in changing the location of a function definition inside the file. Therefore, it is important to find a more meaningful way to analyze source code differences that can express more than naive text differences. The last CVS command that we are going to discuss is CVS update, which is given a revision number, and it synchronizes the local working copy with that revision, so that the developer can control which revision resides in his/her local working copy.

2.2 Bug Reporting Tools

Many open source software projects utilize bug tracking tools as a means of reporting software bugs, assigning them to developers, and monitor their state. Many of bug repositories store the information about bugs and their state in a public database that is accessible to all users through a web interface. Therefore, bug tracking tools in that sense are also a means of interaction between developers and the user community that can be geographically distributed. Users report bugs they find in the software, and discuss open issues with the developers.

When a new bug is reported, the bug is assigned a bug ID by the system, and the bug reporter is allowed to submit an elaborate description of the bug, so that it can be reproduced by the developer who will fix this bug. Information is recorded in the bug report about who the reporter is, the creation time, the component, the operating system and the version. In addition to this information that is collected at bug creation time, there is also information that occurs over the life time of the bug, like the developer to whom the bug is assigned, other people that will be on the communication list when discussions about the bug take place, and the state of the bug, whether still pending, fixed, or cannot be resolved.

When a bug report is submitted to the bug repository, its status is set to NEW. The bug then is examined by a bug triager, who assigns the bug to the appropriate developer, and the bug status is then set to ASSIGNED. When the bug is fixed, its status is set to FIXED. When the bug is found to be a duplicate of another existing bug that has been already reported, this bug is reported as DUPLICATE. If the bug cannot be fixed, it is marked as WONTFIX. Figure 1 shows the web interface for the Eclipse project Bugzilla repository for Bug ID 178190 as an example.

3. CVS Information Extraction

In this section we describe how to obtain structured information from CVS that describes change history on both the file level and method level. File level changes are supported by CVS through the command "CVS log". The output of this command can then be processed and stored in a database relation that has the fields (file name, revision id, developer name, date of change, time of change, number of lines changed). On the other hand, CVS does not support a direct way for finding method level changes. Therefore, special techniques have to be employed to obtain this kind of information. Note that the command "CVS diff" will not help much, because it returns only text differences between two files. It actually treats two files as sequences of characters, even if they contain source code. Therefore, in the following subsection, we will discuss our approaches for finding function-level differences in source code, and then show our proposed approach.

3.1 Method-level difference analysis

Java fact extractor [4] is a software tool that has the capability of parsing both the class files and Java source files in order to extract information about class and method signatures. Extracting information from a Java source file involves implementing almost

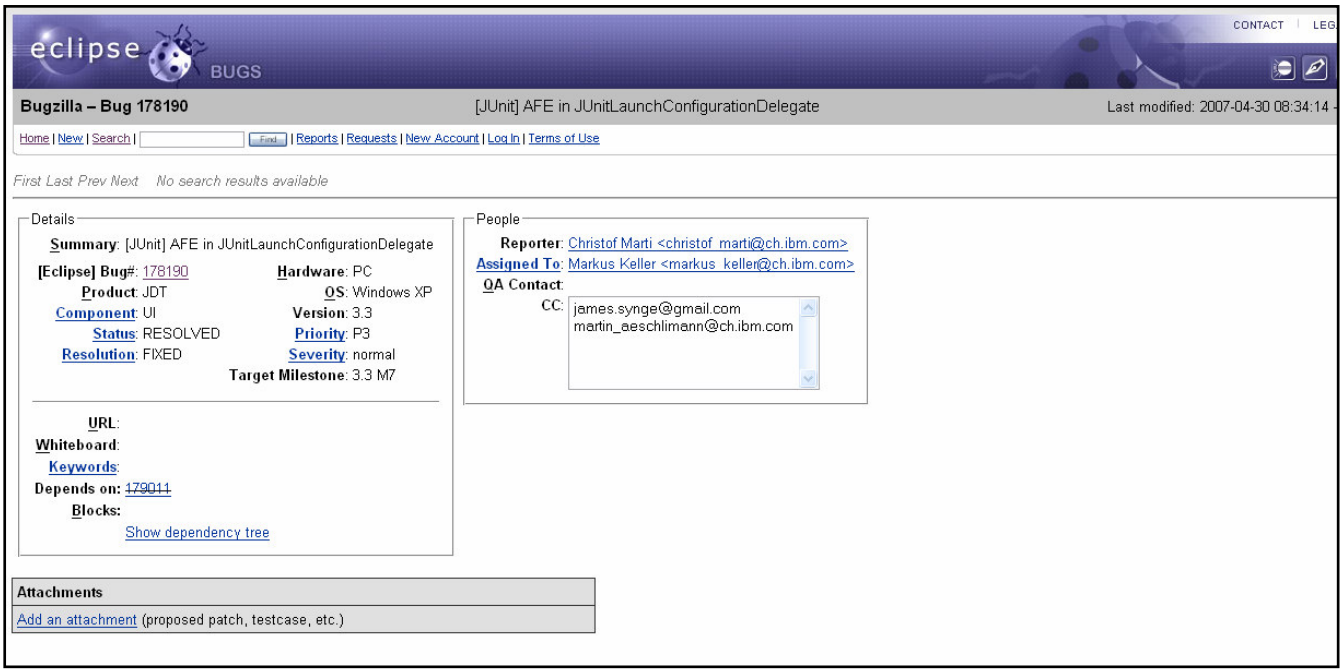

Figure 1: Bugzilla bug report example for Eclipse

all the functionality of a Java source compiler. It also involves completely understanding the language semantics. This scheme works very well to gather course grained information about the class file, its methods modifiers and member variables. However, it fails to capture information from the method bodies, which is an essential requirement in our analysis. Moreover, Java fact extractor is not a differencing tool, which implies that extra steps are required in order to compare different outputs of Java fact extractor.

JDiff [5] is an open source differencing tool that generates HTML reports of all the packages, classes, constructors, methods, and fields which have been removed, added or changed, when two APIs are compared. This is very useful for describing exactly what has changed between two releases of a product. Only the API (Application Programming Interface) of each version is compared. Therefore, like Java Fact Extractor, it does not support detecting differences on the level of method bodies, and will report methods that have the same signatures as identical, even if they implement different functionalities.

Eclipse Compare plugin is a tool that ships with the Eclipse IDE. It is used to find structural differences between two files. That means that it is capable of finding differences on different levels, i.e. classes, constructors, fields and methods. Since Eclipse is an open source project, the Compare module itself can be obtained and used in isolation of Eclipse to compare Java files. An advantage of this tool is that it focuses more on structural differences. That means that changing method location inside the file will not affect the comparison. However, a drawback of using this tool is that it pays attention to source code formatting differences. For example, an "if" statement that is written on one line will be different from the same "if" statement that is written on two lines.

A technique for supporting source code difference analysis was proposed by Maletic et al [6]. In this technique, source code files are converted to the srcML language [7]. The representation, srcML, is an XML format that explicitly embeds abstract syntax within the source code while preserving the documentary structure as dictated by the developer. A drawback of using such an approach is that the srcML representation preservers the documentary structure of the source code. Therefore, it will be affected by many minor and unimportant changes, like whitespaces, whether composite statements are written on one line or more, whether comments are written in the backslash notation or in the star notation, and so on.

3.2 Proposed approach for method level changes

The technique that we propose for solving our problem is by using an XML representation of the source code files; however, by employing compiler output information, rather than source code text information. Therefore, we used the JavaML [8] language to represent the source code files we have, and used the tool Java2XML [9] to perform the conversion. Using this approach has many advantages. It is not sensitive to source code "format", or the location of functions inside the source code. Furthermore, comments can be easily ignored. Also, using XML as a representation in general allows performing XSLT transformations and XPATH queries on the output XML output.

Once source files are converted to XML, XPATH can be used to query the XML output for the contents of the <method> tag that represents methods information. For CVS analysis purposes, this approach can be applied for each two consecutive versions of a file and then the information can be stored in a database relation that has the fields (file name, revision number, class name, different method).

By applying the approaches mentioned above for both file level and method level differences on the Eclipse CVS repository for the JDT module, we found that the repository contains about 200,000 different revisions (and each revision has a group of method differences as well). Therefore, comparing pairs of revisions by converting each revision to XML first would take a very long time on such a big dataset. Therefore, we focused only on the changes done during the year 2007. This reduced the dataset size, so that it had 2871 revisions totaling 5978 method differences.

4. Network Construction

In order to be able to use network analysis approaches on the information extracted from CVS, a network that expresses the relationship between developers and the methods, and also the methods and themselves should be constructed. Therefore, we constructed a network that has the project developers and all the methods in the source code as nodes. Such kind of networks that contains more than one node type is called multimodal networks. In order to construct the network links, we use the information extracted from CVS, to link each developer with the methods he/she changes. We also link methods that change together in the same time to represent relationship between methods. When we say that we group methods based on the change time and the developer, we mean that methods that have been changed on the same day by the same developer are considered related to each others and we link them together in the constructed network.

5. Network Analysis

In this section we section we discuss some of the results we observed from analyzing the network constructed as described in the previous section. We first start by developer importance ranking, then by method importance ranking and finally, we show a way for visualizing the information extracted and extracting useful information from the visualization.

5.1 Developer Importance Ranking

A measure of importance that is usually used in network analysis is *betweenness centrality* [10]. Betweenness centrality is a measure of a node within a graph. Nodes that occur on many shortest paths between other nodes have higher betweenness than those that do not.

For a graph $G: = (V,E)$ with n nodes, the betweenness $CB(v)$ for node v is:

$$C_B(v) = \sum_{\substack{s \neq v \neq t \in V \\ s \neq t}} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where σ_{st} is the number of shortest geodesic paths from s to t, and $\sigma_{st}(v)$ is the number of shortest geodesic paths from s to t that pass through a node v. This may be normalised by dividing through by the number of pairs of nodes not including v, which is $(n - 1)(n - 2)$.

Calculating the betweenness centrality of all the vertices in a graph involves calculating the shortest paths between all pairs of vertices on a graph. This takes $\Theta(V^3)$ time with the Floyd–Warshall algorithm. On a sparse graph, Johnson's algorithm may be more efficient, taking $O(V^2 \log V + VE)$ time.

Therefore, we rank developers according to the betweenness centrality measure. Such a measure should reveal information regarding which developers are more involved in the JDT development more than others, and which developers work on broad topics and which work on specific ones. We show the rankings in Table 1 along with the score of each developer (the ordering is row-major). Figure 2 shows the distribution of developer scores. It can be observed that few developers have very high centrality, with large gaps between them, then the centrality drops very sharply with less gaps.

In order to validate these results, we contacted the developers themselves, and asked them questions about their role in the software, and how would they rank themselves among all the 14 developers in terms of their overall interaction and knowledge of different project pieces. The developer who was ranked second by our measure, responded that he would reply as soon as he gets some time, because he was in a delivery rush (pretty busy!). One developer ranked the UI module team developers according to their interaction with the entire project, and mentioned that the ranking is maeschli , dmegert , mkeller and bbaumgart, which is the same ranking like ours, except for mkeller's rank. The least ranked developer in our measure, responded by that he did not want to answer the question, when we asked him about his belief of his rank.

Developer	Score	Developer	Score
daudel	5899005	oliviert	4351020
ffusier	2946859	wharley	1394260
jeromel	1359807	pmulet	834203.4
mdaniel	387839.4	kent	248123.7
erjodet	237345.3	jgarms	48562.5
mkeller	42813.29	maeschli	28059.27
dmegert	13289	bbaumgart	330.25

Table 1. Developer centrality measure

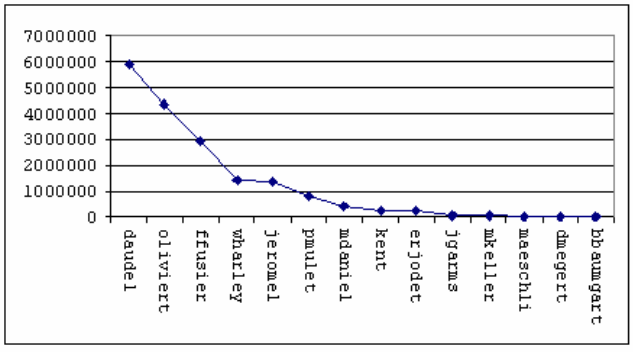

Figure 2: Developer centrality distribution

5.2 Method Importance Ranking

As we ranked developers according to their centrality, we can also rank methods according to their centrality. Such a ranking would reveal information about which methods are "*broker methods*", or methods that connect different parts of the program together. Table 2 shows the names of the 20 highest ranked methods, and

Figure 2 shows the distribution of their centrality score. We can observe two interesting results. First, 19 out of the 20 methods are *testing* methods. This result coincides with the proposition we have just mentioned, regarding the interpretation of method centrality, where we stated that central methods are the broker methods that connect different parts of the program together. Actually, this is exactly what testing suits are doing [11, 12, 13, 14]. Good testing suits are the ones that achieve higher code coverage, and, hence, have relations to many of other program functionalities. A future study can look at how to achieve test prioritization using these ranking results. The second interesting result is that the distribution of the ranks follow a similar pattern to the distribution of the developer ranks, where there are few methods at the top which are very central, followed by other methods that are much less central than the higher ones. However, the degradation is not as sharp as it is in the case with developer rankings.

Method	Rank
ResolveTests.testDuplicateTypeDeclaration7	1
JavadocTypeCompletionModelTest.test024	2
CompletionTests2.testChangeInternalJar	3
CompletionTests2.testBug91772	4
JavadocPackageCompletionModelTest.test025	5
JavadocMethodCompletionModelTest.test139	6
JavadocPackageCompletionModelTest.test031	7
JavadocPackageCompletionModelTest.test024	8
JavadocMethodCompletionModelTest.test038	9
ASTConverterTest2.test0607	10
TestUtils.convertToIndependantLineDelimiter	11
CompletionParserTest2.test0156_Method	12
JavadocTest_1_5._testBug209936	13
ASTConverterTest2.test0608	14
BuildpathTests.testMissingLibrary2	15
BuildpathTests._testMissingLibrary3	16
BuildpathTests._testMissingLibrary4	17
ErrorsTests.test0104	18
InnerEmulationTest.test125	19
VariableElementImpl.hides	20

Table 2. First 20 methods with highest betweenness centrality

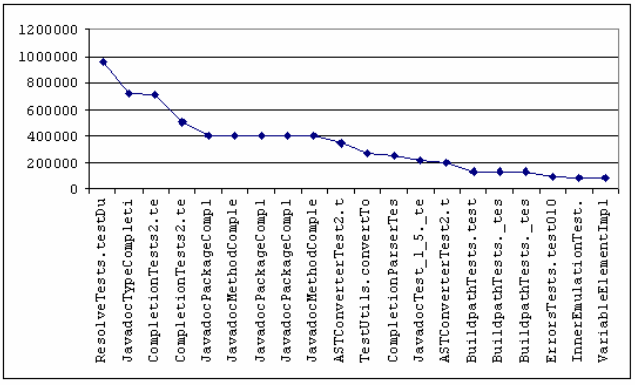

Figure 3: Method centrality distribution

5.3 Visualization

Having constructed a network, a useful way of analyzing it is simply by visualizing this network. However, it becomes prohibitive to visualize networks when the size of the network becomes quite large, like the case under study in this paper. Therefore, it becomes important to find a way to summarize the information in the network, and come out of a smaller network that conveys the most important concepts that exist in the original network. One way of summarizing networks is by filtering out unimportant nodes from it [15]. Therefore, we used the betweenness centrality measure, and removed low ranked nodes from the graph, so that the information about important nodes can be captured quickly. Figures 4 and 5 show two cases where we show only the network containing only the most important 20 and 100 nodes respectively. We removed relationships between methods to aid clearer visualization of links between methods and developers, and hence, focus on information like developer collaboration information. We also used node ranks rather than names to have an idea about the ranks of the nodes collaborating together. We can see in Figure 4 that developers D1 and D2 are collaborating, and also we can see the methods they are collaborating on. In Figure 5, we can see more collaborations than the ones observed in Figure 4. Some examples are between D2 and all of D6, D7, D3, D1,D4, and also between D3 and both of D6 and D5.

6. Link Prediction

As we mentioned earlier, we use link prediction to predict the relationships between developers and the methods in the source code. Link prediction [16] is a set of techniques in network analysis that predict future link formation in a network, given the network structure at the present time. Most of the techniques for link prediction leverage the network structure information to find the predictions. All the methods assign a connection weight score(x, y) to pairs of nodes x and y, based on the input graph, and then produce a ranked list in decreasing order of score(x, y). Thus, they can be viewed as computing a measure of proximity or "similarity" between nodes x and y, relative to the network topology.

There are several methods for calculating the node proximity. The simplest approach to find score of two nodes x, y is to calculate the shortest path between x and y. However, since we prefer higher scores, then we use the negated length of the shortest path. But in this approach, all links that share only one neighbor will

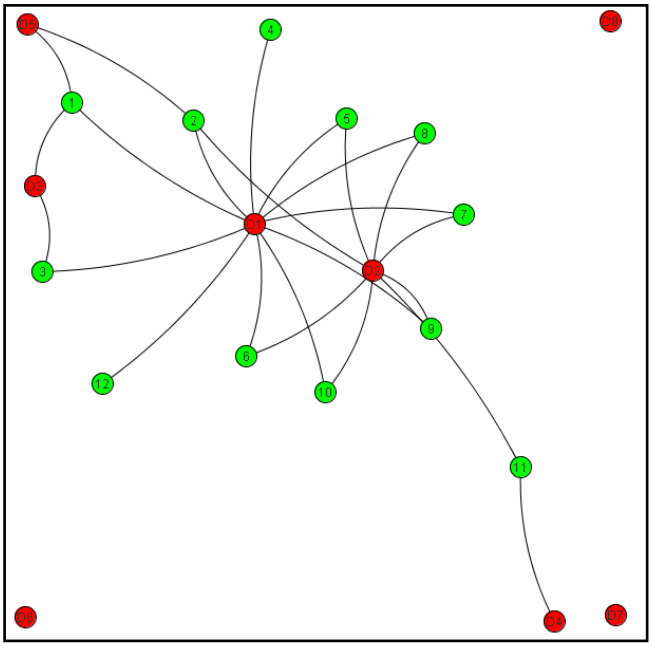

Figure 4: Collaboration graph for the top most 20 important nodes

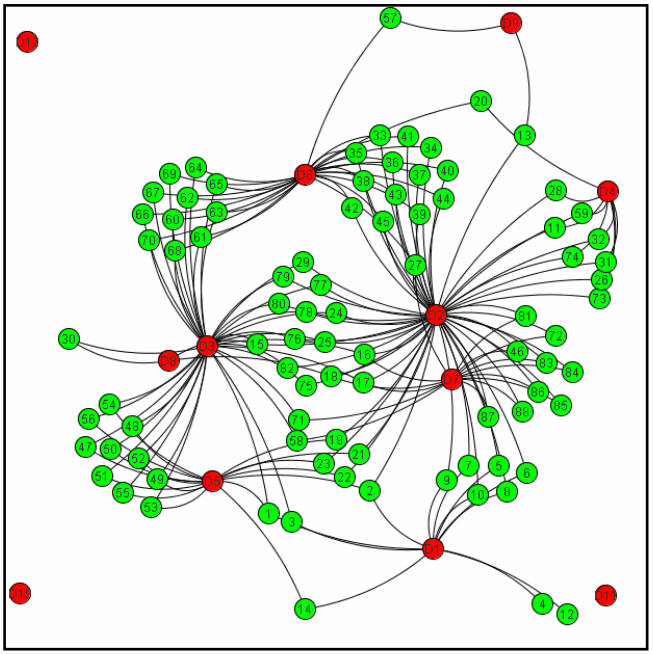

Figure 5: Collaboration graph for the top most 100 important nodes

have higher scores than others. Newman [17] studied the problem of link prediction in the context of scientific collaboration, and he found that the probability of scientists collaborating together increases with the number of other collaborators they have in common. Therefore, he proposed the scoring function:

$$score(x, y) = |\Gamma(x) \cap \Gamma(y)|$$

where $\Gamma(x)$ are the neighbors of node x. Adamic/Adar measure [18] is a measure that counts the common neighbors as well; however, it gives more weight to rare neighbors that are shared with only few other nodes. Therefore, the scoring function is:

$$score(x, y) = \sum_{z \in \Gamma(x) \cap \Gamma(y)} \frac{1}{log|\Gamma(y)|}$$

Another measure by Newman [17] is preferential attachment, which is again in the context of scientific collaboration, and states that the probability of co-authorship of x and y is correlated with the product of the number of collaborators of x and y. The scoring function for that measure is:

$$score(x, y) = |\Gamma(x)|.|\Gamma(y)|$$

Other methods for link prediction include Hitting Time, Rooted PageRank, SimRank [19], Unseen Bigrams, and clustering.

The last link prediction measure that we will discuss is the Katz measure [20], which we used for our link prediction purposes. Katz defines a measure that directly sums over the collection of paths between two nodes x and y, exponentially damped by length to count short paths more heavily. This leads to the measure:

$$score(x, y) = \sum_{l=1}^{\infty} \beta^l . |paths_{x,y}^{<l>}|$$

where $paths_{x,y}^{\langle l \rangle}$ is the set of all length l paths from x to y. (A very small β yields predictions much like common neighbors, since paths of length three or more contribute very little to the summation.) An interesting property of the Katz measure is that the matrix of scores is given by $(I - \beta M)^{-1} - I$, where M is the adjacency matrix of the graph. However, in order for the Katz measure to converge, β should be less than the reciprocal of the largest Eigen value of the adjacency matrix. In our Eclipse CVS network, by computing the Eigen values of the adjacency matrix, we found that the largest Eigen value is 54.2. Therefore, we set β to 0.018, which is the maximum beta less than the reciprocal of the largest Eigen value.

Having obtained the link prediction matrix, we can sort the scores of developers for each method, by extracting the sub matrix *scores*(*x*,*y*) where $x \in$ methods and $y \in$ developers. We will call this matrix *canFixMethod*(*x*,*y*). Therefore, for each $x \in$ methods, we sort the vector *canFixMethod*(*x*) to obtain a ranking on the developers who can work on or fix this method.

7. Who should catch this Exception?

Given an exception that is reported in a Bugzilla bug, we try to match this exception with a developer who can fix it. For that purpose, we employ ranking information obtained by link prediction as discussed in the previous section to calculate an overall ranking for the developers, with respect to the whole exception. Since the exception is a trace of methods that were active in the call stack when the exception took place, it is reasonable to obtain the ranking based on the average developer score over all the functions that show up in the exception trace. Therefore, the score *canFixException*(*z*,*y*) of a developer y to fix an exception z is:

$$canFixException(z, y) = \frac{\sum_{i=1}^{n} canFixMethod(m_i, y)}{n}$$

where the methods m_1, m_2 … m_n are the methods that appear in the exception z stack trace. By sorting the vector *canFixException*(*z*) we obtain a ranking on developers who can this exception.

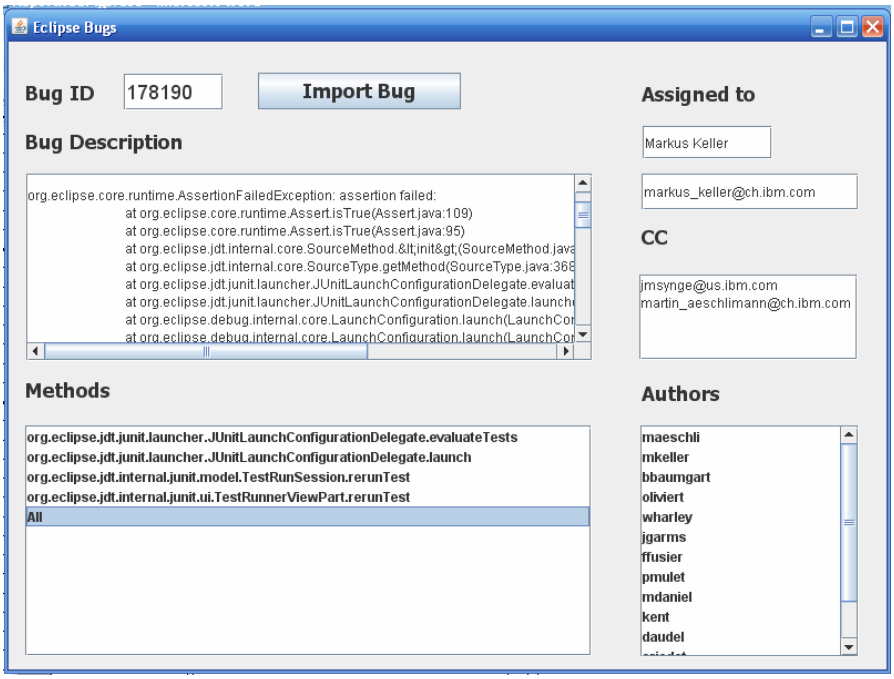

Figure 6: Screen shot of the proposed tool

8. Evaluation

Figure 6 shows an example of using the proposed tool. In the top-left textbox, the user inputs a Bug ID that she wants to find recommendations for. To the right, if the bug was already assigned to a developer, the developer's name and email are shown, and also all other the information about other people that were on the communication list during the discussions about that bug. In the bottom half of the user window, appears the methods that the tool can provide a ranking for, which are basically any methods that exist in the network constructed, and also one more entry that expresses the overall exception recommendation. As we can see in the figure, the user asks for recommendation for the bug report 178190. This is the same bug shown in Figure 1. The tool shows the bug description, and the person who was assigned that bug, Markus Keller, and people who were on the communication list, J M Synge, and Martin Aeschlimann. In the second half, the user is allowed to select the method that she is interested in seeing its individual rankings, and also the user is allowed to see the overall ranking based on all the methods. This should be a ranking for people that can work on that bug. As we can see from the figure, the tool names Martin Aeschlimann as the first recommendation, who was listed on the CC list, and Markus Keller as the second recommendation, who was actually assigned the bug. Furthermore, the tool recommends other developers in the order of their ranks so that they can be picked subsequently if needed.

We collected from Bugzilla all the bugs that took place over the year 2007. Out of them, we found the bugs whose any of their methods was encountered in our social network analysis. There are 16 bugs of this kind. By computing the average ranking as recommended by our tool for the developers who actually fixed these bugs, we found that the average ranking is 2.4, which indicates that the developers who actually solved the bugs were ranked pretty high by our tool. By considering both ranking of the developers who fixed the bug and the developers on the CC list, the average rank of them became 2.9. The reason for the average increase is that now there are multiple persons who are working on a single bug. Therefore, in addition to the person that is ranked number 1, other persons will take ranks more than 1, and may be more, according to the number of people who were actually in both the "Assingned to" and "CC" list.

9. Related Work

Three types of approaches have been used to recommend experts for a software development project: heuristic-based (e.g., [22]), social network-based (e.g., [25]) and machine learning-based (e.g., [26]).

Heuristic-based approaches apply heuristics measures to quantify experience. Some approaches require users to maintain profiles that describe their area of expertise or organizational position (i.e., [21]). However, as expected, it is difficult to keep such profiles up-to-date. Other heuristic-based expertise recommenders are based solely on data extracted from the archives of the software development. The Expertise Locator system [22] uses file dependency matrix that keeps track of how many times pairs of files are changed together, as well as file authorship matrix, that keeps track of how many times developers change different files, to come up with the experience matrix, that shows how much two developers can benefit from each others. The Expertise Browser (ExB), for example, uses the concept of experience atoms (EA), which are basic units of experience, as the basis for recommending experts [23]. Experience atoms are found by mining software repositories for the changes along with their associated information like the developer name, the file containing a modification, the technology used, the purpose of the change and/or the release of the software. A simple counting of experience atoms for each domain in question is then used to

determine the experience in that area. As another example, the Expertise Recommender (ER) [21] was proposed as method for finding experts based on the developers change history. The authors assume that the most appropriate developer for a module is the one who changed it last. Girba et. al. [24] used line-level approach for locating experts. The authors assumed that each developer has an amount of experience proportional to the number of lines she changed. However, this measure is not always accurate and not indicative of the experience. For example, some changes are done in a batch manner, or sometimes just add few comments. Social network approaches have been addressed as well. The approaches describe relationships between developers built using data mined from the system development. For example, in [25] the authors are studying the open source project development phenomena using a social network approach, where they link two developers if the collaborate on the same project. They find that this collaboration network follows a power law distribution, and there are few developers who are working on multiple projects. Machine learning-based approaches used text categorization techniques to characterize find developers who should fix a bug [26]. In these approaches, existing bug reports along with their bug assignments are given to a text categorization algorithm, so that future bug reports can be predicted for their appropriate developer who should handle them. This approach is different than ours in that we do not train our system on existing but reports, rather, we train it on information that already exists in the software project repository.

10. Conclusions

In this report we have studied approaches for applying network analysis methods on information extracted from source code repositories. First, we have seen an approach for how to get semantic differences between different versions of the source code. Then we utilized the information extracted regarding method level differences to construct a network of relationships between the developers and the methods on one hand, and the methods and themselves on the other hand. We also showed techniques for analyzing this network by ranking, visualization and link prediction. Lastly, we showed how link prediction can be employed to predict who the developers are that can fix an exception. Experimental evaluation showed that network analysis is a powerful tool to promote the understanding of software projects, and that link prediction approaches are effective methods for finding experts. Future directions include validating the developer rankings by contacting the developers themselves, and exploring their work nature. Also, we will consider using method ranking to aid test prioritization. Furthermore, from the network construction point of view, we will consider the enriching the network's link structure by adding links between methods that call each others, or methods that are in the same class. Moreover, an improvement to the accuracy can be preformed by making use of previous bug reports by linking them to the developers who actually fixed them, and predict developers who should fix new bugs. From the computational point of view, we will consider expanding the system so that it can incrementally be updated when new data comes to the scene. As we have seen, the data sets are huge, and doing the entire analysis every time new transactions take place will be prohibitive.

11. References

[1] CVS Project Homepage http://www.nongnu.org/cvs/

[2] Bugzilla Project Homepage http://www.bugzilla.org/

[3] Wikipedia article http://en.wikipedia.org/wiki/Revision_control

[4] Java Fact Extractor http://www.swag.uwaterloo.ca/javex/index.html

[5] JDIFF Tool Project Homepage http://javadiff.sourceforge.net/

[6] Jonathan I. Maletic, Michael L. Collard: Supporting Source Code Difference Analysis. ICSM 2004: 210-219

[7] M. Collard. Addressing source code using srcml. In IEEE International Workshop on Program Comprehension Working Session: Textual Views of Source Code to Support Comprehension (IWPC'05), 2005

[8] G.J. Badros, "JavaML: A markup language for Java source code," Proc. Int'l WWW Conference, May 2000.

[9] https://java2xml.dev.java.net/

[10] M. E. J. Newman, The structure and function of complex networks, SIAM Review 45 , 167-256 (2003).

[11] Coward, P. "A review of Software Testing", Information and Software Technology, Vol. 30, no. 3 April 1988, pp. 189-198.

[12] Laprie, 1.-C., Dependability: Base Concepts and Terminology, vol. 5 in the Series on Dependable Computing and Fault-Tolerant Systems, Springer-Verlag, Austria, 1992.

[13] Howden, W., Reliability of the Path Analysis Testing Strategy, IEEE Trans. Software Eng. SE-2 (1976), pp. 208-215.

[14] Howden, W., Functional Program Testing and Analysis, McGraw-Hill, 1987M. E. J. Newman, The structure and function of complex networks, SIAM Review 45 , 167-256 (2003).

[15] Adam Perer, Ben Shneiderman: Balancing Systematic and Flexible Exploration of Social Networks. IEEE Transactions on Visualization and Computer Graphics (InfoVis 2006). 12(5): 693-700 (2006)

[16] David Liben-Nowell, Jon M. Kleinberg: The link prediction problem for social networks. CIKM 2003: 556-559

[17] M. E. J. Newman. The structure of scientifc collaboration networks. Proceedings of the National Academy of Sciences USA, 98:404-409, 2001.

[18] Lada A. Adamic and Eytan Adar. Friends and neighbors on the web. Social Networks, 25(3):211{230, July 2003.

[19] Glen Jeh and Jennifer Widom. SimRank: A measure of structural-context similarity. In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 2002.

[20] Leo Katz. A new status index derived from sociometric analysis. Psychometrika, 18(1):39-43, March 1953.

[21] D. W. Mcdonald and M. S. Ackerman. Expertise recommender: a flexible recommendation system and architecture. In Proc. of CSCW, pages 231–240, 2000.

[22] Shawn Minto and Gail C. Murphy. Recommending emergent teams. MSR 2007.

[23] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In Proc. of ICSE, pages 503–512, 2002.

[24] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In Proc. of IWPSE, pages 113–122, 2005.

[25] G. Madey, Freeh, V., and Tynan, R. "The Open Source Software Development Phenomenon: An Analysis Based on Social Network Theory". Americas Conference on Information Systems (AMCIS2002). Dallas, TX, 2002. pp. 1806-1813

[26] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In Proc. of ICSE, pages 361–370, 2006.