Predicting Buggy Changes Inside an Integrated Development Environment

Janaki T. Madhavan
jmadhava@ucsc.edu

E. James Whitehead, Jr.
ejw@soe.ucsc.edu

Department of Computer Science
University of California, Santa Cruz
Santa Cruz, CA 95064

ABSTRACT

We present a tool that predicts whether the software under development inside an IDE has a bug. An IDE plugin performs this prediction, using the Change Classification technique to classify source code changes as buggy or clean during the editing session. Change Classification uses Support Vector Machines (SVM), a machine learning classifier algorithm, to classify changes to projects mined from their configuration management repository. This technique, besides being language independent and relatively accurate, can (a) classify a change immediately upon its completion and (b) use features extracted solely from the change delta (added, deleted) and the source code to predict buggy changes. Thus, integrating change classification within an IDE can predict potential bugs in the software as the developer edits the source code, ideally reducing the amount of time spent on fixing bugs later. To this end, we have developed a Change Classification plugin for Eclipse based on client-server architecture, described in this paper.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement—*version control*; D.2.6 [**Software Engineering**]: Programming Environments—*Integrated environments, Programmer workbench*

General Terms

Algorithms, Measurement, Experimentation

Keywords

Bug prediction, integrated development environments

1. INTRODUCTION

Bug prediction has justifiably been an important field of research in software engineering, as the amount of effort, time and expenditure spent on fixing bugs is usually quite high. As a result of this research, many techniques have emerged for detecting and predicting bugs in software. Many bug detection techniques are based on static and dynamic analysis of programs [5, 11, 10]. Some techniques analyze software metrics or a project's change history to identify problematic modules [14, 17, 22]. Others use classification or regression algorithms with features such as complexity metrics, cumulative change and bug counts to predict risky entities [12, 4, 14, 17, 23, 21]. Some techniques use statistical analysis and machine learning techniques to distinguish between failing and passing program executions [3, 13, 9].

Change Classification [19] uses Support Vector Machines (SVM), a set of related supervised learning methods that are particularly effective in text classifications. Change Classification learns from the change history of a project to classify any future change as clean or buggy. Unlike other techniques based on classifier algorithms, here the features are extracted from all sources of information of a project, viz., change log messages, change delta and the source code, complexity metrics, author, time and day of submission, file and directory names, etc, to create a corpus for the classifier to train on. This leads to classifications of relatively high accuracy. In addition, this technique is language independent and predictions can be done immediately at the level of individual source code changes.

In this paper we describe a tool in the form of an Eclipse plugin that uses Change Classification to effectively classify changes as buggy or clean during the editing session itself. This is possible because the features extracted solely from the change delta and the source code, collectively known as AND (A:added delta, N:new revision, D:deleted delta) are seen to have relatively high accuracy in classifying changes. Thus, the developer has the opportunity to take measures immediately. These measures could include getting information about the change by (a) using his/her memory of any similar change that resulted in a bug, (b) describing the change to other developers in the group to enquire about any similar bug introducing change they might have made, (c) manually going through the revisions using the change logs to look for similar changes; this can be done only if the project is not very large and does not contain many revisions, (d) obtaining the bug introducing changes using tools like Kenyon [2] and finding the changes most similar to the current one using relevant similarity measures, (e) performing a software inspection on code that was flagged as buggy before it is checked in or (f) run one or more static analysis tools to detect bugs (knowing that the high false positive rate of such tools will be acceptable since there is a high like-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA '07 Montréal, Québec, Canada
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

lihood of there being an actual bug). The developer could then make an informed decision about the change. These measures will necessarily increase time spent in coding, but will save time in the future since the latent bug will be discovered immediately, without requiring time to isolate the bug, have a developer refresh their memory on the code, and then implement the fix. Using Change Classification, less time is required to isolate the bug and implement the fix.

The rest of the paper is organized as follows: In Section 2, we discuss the related work in bug detection and prediction. In Section 3, we discuss Change Classification in greater detail. In Section 4, we describe our Change Classification Eclipse plugin, its architecture and implementation. Section 5 describes the future work and Section 6 concludes.

2. RELATED WORK

Many automated bug finding tools have been proposed and are in wide use [15, 11, 7, 1, 20]. They are good at finding commonly known bugs but do not detect project-specific ones. Some tools address this problem, for example, BugMem [18], by using the projects' change history to identify project-specific bugs patterns.

Some bug finding tools have been incorporated into Eclipse as plugins, for example, FindBugs [15] and Checklipse [20]. They use lightweight static analysis techniques to detect bugs in Java code. The AMPLE [8] plugin locates failure causes in Java programs by comparing the method call sequences of passing and failing program runs. There are plugins that study the impact of a change request on other work products to reduce potential bugs [6, 24, 28]. The HATARI plugin relates the version history of projects to a bug database to find risky locations in the code [26].

The Change Classification plugin, like HATARI, uses the software's version history to predict bugs. It is based on Change Classification [19], a bug prediction technique that uses machine learning classifier algorithms to classify changes made to the source code as buggy or clean. Also, besides being language independent, our plugin is different from others in that it can classify changes as the developer edits. Therefore, the developer has immediate feedback on any potential errors they have made.

3. CHANGE CLASSIFICATION

Change Classification [19] is based on Support Vector Machines (SVM) [16], a set of related supervised learning algorithms, that are effective in a wide range of text classifications. An SVM is a discriminative model which tries to find the maximum margin hyperplane between the different classes. SVM is first trained on labeled data instances, which are known to belong to one class or the other. Subsequently, unlabeled instances can be classified to belong to one class or the other by the SVM. In Change Classification, the binary classification is between a clean change and a buggy change.

The Change Classification project has created training sets for 12 mature open source projects [19]. They extract file level changes from the project histories of the 12 projects using Kenyon [2]. They, then identify the changes that introduce bugs and those, which don't. The techniques used are described in [19]. Every term in the added delta (A), new source code (N), deleted delta (D), change log and file name is used as features. They are extracted by using Bag of Words (BOW) [25] and its variants BOW+ and BOW++ [19]. The author name, commit hour and day, cumulative change and bug counts, length of the change log, LOC of the change log, new revision source code are all used as features. In addition, complexity metrics are also computed to be used as features. With these features, they create a corpus of labeled changes with a set of features associated with each change; this is used as the training set for the SVMs.

The performance of Change Classification is evaluated using 10-fold cross-validation method. Its accuracy is seen to range between 63% and 92% in the projects analyzed by the developers of this technique. Buggy change recall ranges between 43% and 86%. Buggy change precision is between 44% and 85%. The feature group consisting of the new revision source code (N), the change delta (added (A) and deleted (D)) leads to a relatively high accuracy of prediction.

Therefore, Change Classification predicts bugs at a file level granularity with accuracy at par with the best bug-prediction techniques. This would be an ideal technique to incorporate into an IDE. Since Change Classification learns from past changes, it can only be applied to projects with some revision history. However, it was observed in the Change Classification project that changes could be effectively classified based on a history of about 250 revisions [19].

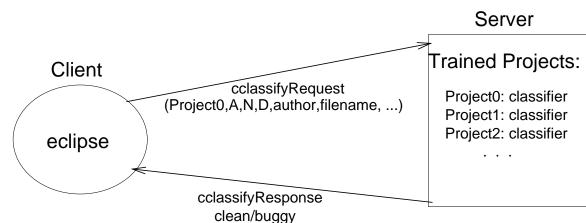

Figure 1: Prototype of the Change Classification plugin.

4. ECLIPSE PLUGIN

The main advantage of having Change Classification as a plugin in a fully featured work environment like Eclipse is that it can give immediate feedback on a developer error during the editing session itself.

4.1 Plug-in Prototype

The Change Classification prototype is that of a client-server system (Fig. 1). The user commands the client (Eclipse) to collect the changes made to the source code since the last time Change Classification ran in that editing session. The client ships the changes to the server, along with the project name, file name, author's name, time and the hour of submitting the change. The server maintains a collection of projects with trained classifiers. If the project is present in this collection, the corresponding classifier classifies the change as buggy or clean. The result is then sent to the client, which then displays it to the user in a human-readable format. If the project is not present in the collection, an appropriate error message is sent to the client.

4.2 Change Classification Client

The client is a plugin that works within Eclipse 3.2 with

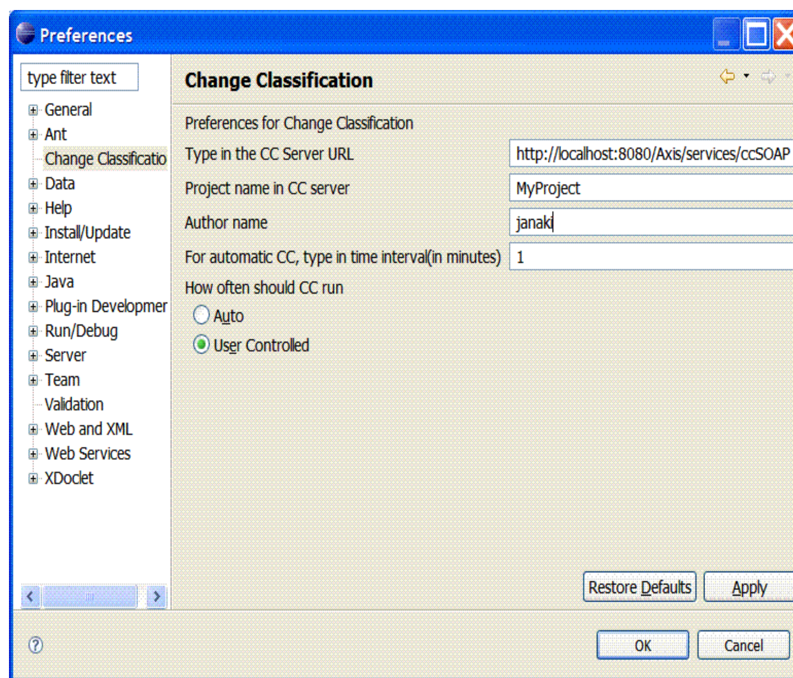

Figure 2: The preference page of Change Classification in Eclipse.

Web Tools Platform [1]. The plugin has a preference page and a view associated with it. The preference page is where the user sets the URL of the server, the project name, his/her name and whether the classification is user-controlled or automatic (Fig. 2). For the latter, the user must specify the interval of classification in minutes in the preference page. The automatic mode can be used if the user knows approximately how long it takes for them to make the changes before committing. Therefore the user can continue working on the code, without stopping to classify the changes. On the other hand, if the user wants the classification to be done only when they want it (for example, exactly before a commit), the user-controlled mode is more appropriate. The default mode of Change Classification is user-controlled.

Clicking on "Change Classification", that appears in the pop-up context menu of the files in the Package explorer view, opens the workbench editor on that file and takes a snapshot of the initial state of the text. If the classification is set to be automatic, a document listener (*TimeDocumentListener*, implemented as a subclass of *IDocumentListener* of the package *org.eclipse.jface.text*) is added to the underlying document (*IDocument*). This listener listens to changes made to the text and at user-specified time intervals, takes snapshots of the current state of the text. On the other hand, if the classification is user-controlled, the user clicks on "cclassification" on the workbench menubar or on the icon of a blue planet in the workbench toolbar (Fig. 3), whenever they want to take snapshots of the text. Then, the differences between subsequent snapshots are obtained using the *RangeDifferencer* class contained in the package *org.eclipse.compare.rangedifferencer*. The added delta, the new source code and the deleted delta (AND) are obtained as simple strings and sent to the server, along with other information like the project and the author names, the time and the day of submitting the change.

The response from the server is a string, which is displayed in the Change Classification view of Eclipse (Fig. 3). Also, Change Classification does not slow down typing as it is executed in a separate thread.

4.3 Change Classification Server

We have implemented the server as an Apache Axis Web service[2], running within Tomcat[3]. The server and the client communicate using SOAP[4]. *CClassifyService* is a document style Web service. The two operations defined for *CClassifyService* are *train* and *cclassify* and the corresponding messages are *trainRequest* and *cclassifyRequest*, respectively. The response messages of both are strings. The message *trainRequest* contains a string, viz., the project name, while *cclassifyRequest* contains 8 strings, viz., AND, names of the file, project and the author along with the hour and day of submitting the request (all these are used as features during classification).

The server maintains a Hashtable of projects, with trained SVM classifiers. The project names act as the keys. The training sets, which trained the classifiers, are present in directories named after the projects. Upon receiving the project name from *trainRequest*, the server checks whether the project is present in its Hashtable. If so, the message is sent to the client indicating the same; else, if there exists a training set for the project, a new SVM classifier is instantiated and trained. The project, with its trained classifier is then added to server's Hashtable of projects. If there is no training set for this project, an appropriate error message is sent back to the server.

[1] www.eclipse.org/webtools/

[2] ws.apache.org/axis

[3] jakarta.apache.org/tomcat

[4] http://www.w3c.org/TR/SOAP

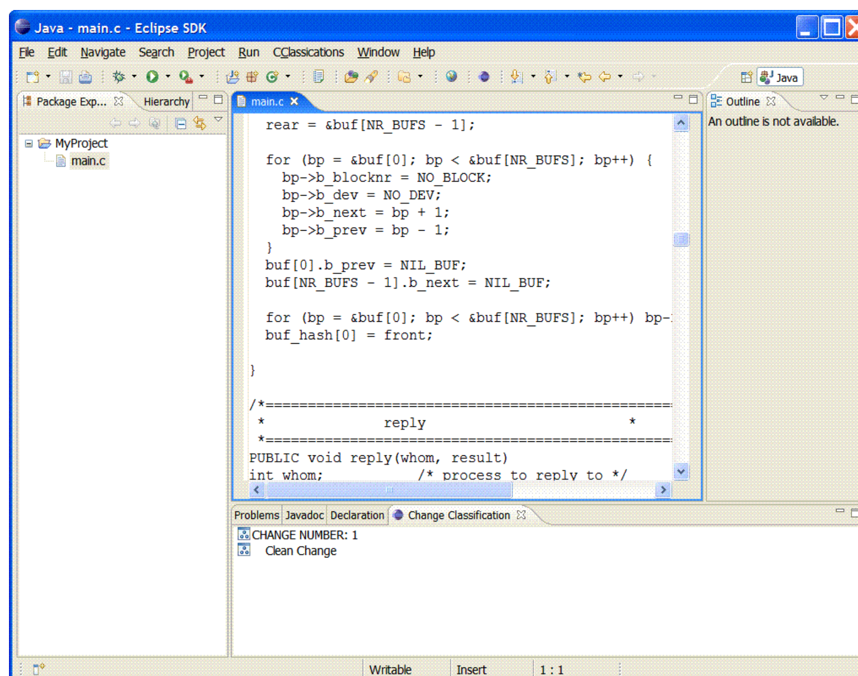

Figure 3: Screenshot of Change Classification in Eclipse. The bottom panel is the Change Classification view, where the results of the classification get displayed.

The *cclassifyRequest* , on the other hand, operates as follows. Upon receiving the message, the server extracts features from AND, author name, file name, hour and day of submission, by using techniques described in [19]. The server gets the trained classifier for the project from its Hashtable (if present) and classifies the instance obtained from the extracted features as buggy or clean. An appropriate message is sent back to Eclipse, where it is displayed in the plugin view (Fig. 3).

The developers of Change Classification [19] use the Weka Toolkit [27] implementation of Support Vector Machines and we do the same. The time taken by the Change Classification server to process a request depends on the number of features in the training set. As the number of features increases, the amount of time taken to classify a change also increases. We classified instances with 300, 3000 and 10800 features and found that the average time taken to process each one was 0.086 seconds, 0.867 seconds and 2 seconds, respectively. Hence, the number of features should be kept in check for a good performance. This can be done by devising a way to select only the "essential" features and by limiting the number of revisions considered while making the training set. It was observed in the Change Classification project that training sets based on as few as 250 revisions are enough to get classification results of relatively high accuracy.

5. FUTURE WORK

A tool that can predict bugs while a developer is writing code is new; hence, we would like to study the following aspects: how convenient it is to use and how do developers react to the change classification. For the former, we would like to collect anecdotes about users' experience with the tool. In particular, we would like to know if the users find the tool easy to use, obtrusive or not obtrusive enough.

The second aspect we would like to study are the developers' reactions to the results of our plugin. The possibilities are that the developer (a) takes the classification seriously most of the time; if so, what are the kind of changes that get ignored, (b) ignores the classifications most of the time due to constraints like time, etc; if so, what kind of changes get taken seriously, (c) takes every classification seriously or (d) ignores every classification. We believe, this study could give us an idea about the "serious" and the "non-serious" changes, as per the developer.

Another issue is that of creating the training sets. For this project, we have used those created by the developers of Change Classification. Since Weka requires this set to be in the ARFF format [27], there is some amount of preprocessing involved. This step could be incorporated in our server itself or could be kept separate in a different server.

6. CONCLUSIONS

In this paper, we have presented a bug-prediction tool that classifies changes made to the source code as buggy or clean during the editing session itself. This is in the form of an easy-to-use Eclipse plugin, based on the Change Classification technique. This technique has been evaluated and found to be very effective in classifying changes. Learning from the past history of changes, our plugin can help the developer find out if the changes they have made to the source code are potentially buggy.

Change Classification can be user-controlled or automatic. The classification doesn't effect the UI responsiveness as it runs in a separate thread. We believe that running the Change Classification plugin in combination with other bug-detection and bug-prediction plugins of Eclipse can potentially save much effort, time and expenditure in the long term.

7. ACKNOWLEDGMENTS

We thank Sunghun Kim for the useful discussions and providing us with the training sets to use for our work.

8. REFERENCES

[1] C. Artho. Jlint - Find bugs in Java programs, 2006.

[2] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey. Facilitating software evolution with Kenyon. In *Proceedings of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005)*, pages 177–186, September 2005.

[3] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behaviour. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04)*, pages 195–205, 2004.

[4] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 480–490, May 2004.

[5] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice & Experience*, 30(7):775–802, June 2000.

[6] G. Canfora and L. Cerulo. Jimpa: An Eclipse plug-in for impact analysis. In *Proceedings of the 10th Conference on Software Maintenance and Reengineering (CSMR 2006)*, pages 341–342, 2006.

[7] T. Copeland. *PMD Applied: Centennial Books*. 2005.

[8] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight bug localization with AMPLE. In *Proceedings of the 6th International Symposium on Automated analysis-driven debugging*, pages 99–104, 2005.

[9] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 339–348, 2000.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Engin.*, 27(2):1–25, February 2001.

[11] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 37, pages 234–245, June 2002.

[12] T. Gyimothy, R.Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Softw. Engin.*, 31(10):897–910, October 2005.

[13] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil. Applying classification techniques to remotely-collected program execution data. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 146–155, 2005.

[14] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 263–272, September 2005.

[15] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Proceedings of the Onward! Track of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 132–136, October 2004.

[16] T. Joachims. Text categorization with support vector machines: Learning with many relevant features. In *Proceedings of ECML-98, 10th European Conference on Machine Learning*, pages 137–142, April 1998.

[17] T. Khoshgoftaar and E. B. Allen. Ordering fault-prone software modules. *Softw. Quality Control J.*, 11(1):19–37, May 2003.

[18] S. Kim, K. Pan, and E. J. Whitehead, Jr. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2006)*, pages 35–45, November 2006.

[19] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or buggy. *IEEE Trans. Softw. Engin.*, in review. Manuscript available at `http://www.cs.ucsc.edu/~ejw/papers/cc.pdf`.

[20] V. Livshits. Turning Eclipse against itself: Finding bugs in eclipse code using lightweight static analysis. Eclipsecon '05 Research Exchange, March 2005.

[21] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Tech. J.*, 5(2):169–180, April-June 2000.

[22] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Engin.*, 31(4):340–355, April 2005.

[23] K. Pan, S. Kim, and E. J. Whitehead, Jr. Bug classification using program slicing metrics. In *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, pages 31–42, September 2006.

[24] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chelsey. Chianti: A tool for change impact analysis of java programs. In *Proceedings of the 19th ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '04)*, pages 432–448, 2004.

[25] S. Scott and S. Matwin. Feature engineering for text classification. In *Proceedings of the 16th International Conference on Machine Learning*, pages 379–388, June 1999.

[26] J. Sliwerski, T. Zimmermann, and A. Zeller. HATARI: Raising risk awareness. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE '05)*, pages 107–110, 2005.

[27] I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Francisco, CA, 2005.

[28] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 563–572, 2004.