Fault-Proneness Estimation and Java Migration: A Preliminary Case Study

Mirco Bianco, Daniel Kaneider, Alberto Sillitti, and Giancarlo Succi

Center for Applied Software Engineering, Free University of Bolzano-Bozen,
Mustergasse - 4 - Via della Mostra
I-39100 Bozen - Bolzano, Italy
{Mirco.Bianco, Daniel.Kaneider, Alberto.Sillitti, Giancarlo.Succi}@unibz.it

Abstract. The paper presents and discusses an industrial case study, where an eight year running software project has been analyzed. We collected about 1000 daily-versions, together with the file version control system, and bug tracking data. This project has been migrated from Java 1.4 to Java 1.5, and visible effects of this migration on the bytecode are presented and discussed. From this case study, we expect to observe the effects on the code size produced by the Java technology migration, and to improve the performances of already existing fault-proneness estimation models. Preliminary results about fault-proneness estimation are shown.

Keywords: Product metrics, process metrics, java migration, fault-proneness estimation.

1 Introduction

Collecting software metrics from an industrial project, is always a good opportunity to learn something new. Doing experiments in the real-world may lead to unexpected conclusions, especially when the whole development context is unknown. Product data together with process data can give a better understanding of the process. In this paper, we present a case study, where software metrics and process metrics are both collected. Using our static code analyzer [14], we gathered data from about 1000 versions of the same software system. This system has been developed by five developers during the last eight years. The collected software metrics are the followings: CK [6] metrics, Source Line of Code (SLOC), and McCabe's Cyclomatic Complexity. In addition, we collected information from the file version control system and from the bug tracking system; we refer to bug tracking data with the term "process metrics". The software project under analysis is written in Java. When the source code has been ported from Java 1.4 to Java 1.5, the project received a major code refactoring. Software metrics result affected.

We traced the bug fixing process using the bug tracking data and the file control system information; this enables us to map the bugs with the corresponding classes. These historical data allow us to build more reliable models for predicting the fault-proneness of Java classes. Moreover, we can observe how the bug trend is influenced by the Java migration.

The analysis of the extracted data has been done using Lagrein [10], SyQL

[3], and Weka[1]. Lagrein is useful for creating dynamic views of the system, it can show how the system grows and evolves; SyQL is helpful to prepare the data sets, especially when we need to join software metrics with process metrics, or rather when we need to map bugs with the corresponding classes. Finally, Weka is indispensable to build and to validate models.

The organization of the rest of the paper is as follows: In the next section, we present the related work. In Section 3, we show how the data has been collected. In Section 4, we show how we have processed the data, and the result of the analysis. Finally, Section 5 draws the conclusions and presents future directions.

2 Related Work

The two main topics involved in this case study are metric-based fault-proneness prediction and software migration. For both, we provide a review of the literature. In Table 1 and 2, a comparison between different case-studies is provided.

Table 1. Related Work on Metric-Based Fault-Proneness Prediction.

Study	Modeling Method	Programming Language	Project size
Basili et al. [1]	LR	C++	180 classes, 8 modules
Briand et al. [4]	MARS	Java	2 projects, 212 classes, 2,707 methods
Cartwright et al. [5]	Linear regression	C++	133,000 LOC, 32 classes
Gyimóthy et al. [9]	LR+ML	C++	Mozilla project, >1,000,000 LOC
Kanmani et al. [11]	NN	C++	200 similar systems, 1,185 classes
Nagappan et al [13]	LR	C++/C#	5 projects, 1,100,000 LOC
Subramanyam et al. [15]	OLS	C++, Java	706 classes
Tomaszewski et al. [16]	Linear regression	-	2 systems, 1800 classes, 1,100,000 LOC
Zhou et al. [17]	ML + SFR	C++	Subset of KC1, NASA project, 145 classes, ~about 40,000 LOC
Our work	-	Java	4,813 classes, 68,666 methods, ~80,000 SLOC

LR = Logistic regression, MARS = Multivariate Adaptive Regression Splines, OLS = Ordinary Least Square regression, NN = Neural Network, ML = Machine Learning, SFR = Severity Fault Ranking

[1] http://www.cs.waikato.ac.nz/ml/weka/

Concerning to metric-based fault-proneness prediction, a lot of work has been already done[1][4][5][9][11][13][15][16][17]. Our project, in comparison to other Java projects [4][15], is larger in terms of number of classes/methods and SLOC (Source Lines Of Code). To build the fault-proneness prediction models, different techniques have been adopted. Machine learning is frequently adopted. Hence, we are going to use this set of techniques to build our bug predictors. Specifically, the machine learning algorithm adopted for this preliminary analysis is the ID3Numerical (Decision Tree); we choose it because in a previous case study [2], it produced models with the highest correctness/ completeness values.

About Java 1.4 – 1.5 migration (see Table 2), we found only one work [7], it treats about the influence of generics and other new language features into Java programming style. In this study, they did not observe the migration of the same project from Java 1.4 to Java 1.5 technology. The quantity of classes analyzed in [7] is considerably larger than our study. Therefore, we are going to replicate the study using two versions (Java 1.4 and 1.5) of the same project, and we expect to reach similar conclusions. We want to remark that the results of the migration are not presented in this paper.

Table 2. Related Work on Java 1.4 – 1.5 Migrations.

	Our work	**Clarke et al. [7]**
Analyzed program versions	995	1
Analyzed projects	1	22
Total analyzed classes	10,716	155,336
Collected metrics	CK based	Based on class characteristics[2]

3 How we collect data

The data collection is performed automatically using promPM [14]. This tool allows us to process a huge quantity of source code/bytecode in a fully automated way. In this paragraph, we start introducing the data collection site, after that, we briefly present the data collection infrastructure adopted in this case study.

3.1 Data Collection Site

The data collection site is inside the borders of the companies that developed the project. The contributors to this project are Pro Data[3] and ASA Software House[4], which started to operate and to gain experience in the South Tyrol area in 1988 and 1989 respectively.

Since the mid-1990s they formed a strategic alliance, and nowadays they provide e commerce desktop applications for different sectors such as service trade, agro-industries and tourism, sharing the same main code base.

[2] Collected with TaxTOOLJ [Babich et al., 2006].
[3] http://www.prodata.it/
[4] http://www.asaon.com/

3.2 Data collection infrastructure

Figure 1 shows the components used to collect data. The Source Code Analysis components ran on a standalone machine. The source code was checked-out from the File control System taking the latest version of each working day.

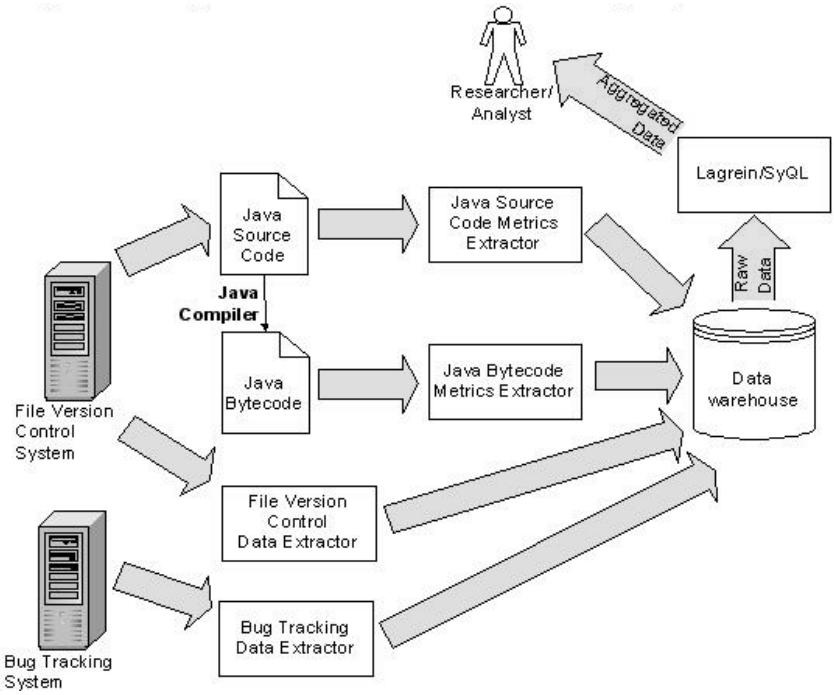

Fig. 1. The System Architecture.

The Java source code metrics extractor is based on an Island Grammar [8][12] parser; the Bytecode metrics extractor uses CKJM[5] to parse the Java bytecode.

The code analyzers store the collected data into the relational data warehouse. The raw data is filtered and aggregated using SyQL, the visualization is entrusted to Lagrein. Information contained into the File Version Control System (files addition, deletion, and modification) is collected using an ad hoc data extractor. This component is specifically designed to extract information from the SVN repository, and it uses the log dump feature available in the most SVN command line clients. There is also an ad hoc component for collecting bug tracking information from the proprietary bug tracking system. Figure 2 shows the collected metrics from the bytecode, we experience that the migration from Java 1.4 to Java 1.5 produces a drastically reduction of the computed Lines of Code (the reader can observe this step at the end of 2007). This happens because Java 1.4 compiler generates synthetic fields into the class scope more often than Java 1.5 compiler. Therefore, we decided to overcome definitively the problem extracting Source Line of Code and the McCabe's Cyclomatic Complexity directly from the source code using a parser written by us. This component has been developed using Island Grammar.

[5] http://www.spinellis.gr/sw/ckjm/

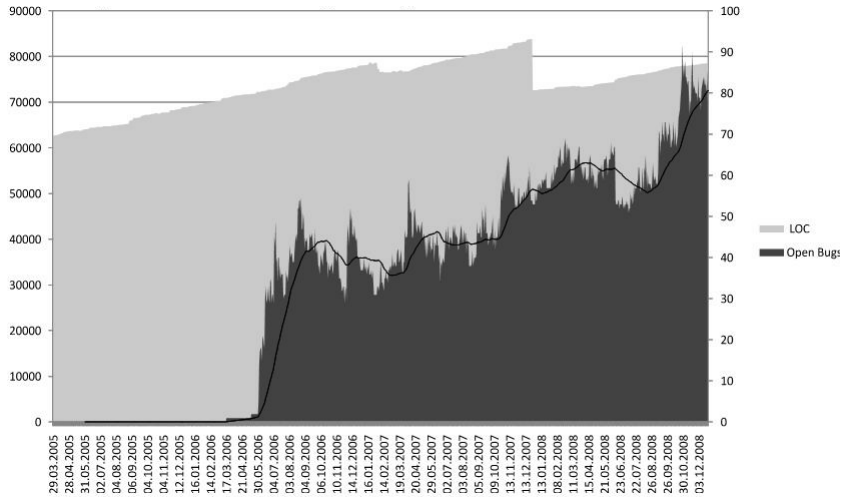

Fig. 2. Preliminary Data Collected from Java bytecode.

4 The Analysis

In this section, we show the preliminary result of our analysis on fault-proneness estimation. The analysis is performed over a 3-year period (2006.06.16 – 2009.02.05). We started from the mid of 2006 because the bug data start to be collected in the second quarter of the same year (see Figure 2). In this analysis, we tried to predict if a class is faulty or not. We consider the extracted software metrics as independent variables, and the faulty flag as dependent variable. To build the training data set, we have combined the following data sources: software metrics, SVN logs, and bugs tracking system. Since the location of Java source code reflects the fully qualified name of each class (e.g. "./src/com/package1/class1.java" contains the definition of the type "com.package1.class1"), we used file system locations to join software metrics with SVN logs data sources. To combine the SVN logs data source and the Bug tracking data source, we used the SVN revision number generated every time a single bug was fixed. This information has been made available from the proprietary bug tracking system. The developers insert this information manually at the end of each bug fix. In this way, we can categorize the faulty and the non-faulty classes.

In the next subsection, we introduce the data set. After that, we present how we have processed the data, and we discuss the obtained results.

4.1 The data set

In this study, we have used the CKJM and the island grammar parser. These components extract 12 different metrics for each class:

- WMC: Weighted Method per Class;
- DIT: Depth of Inheritance Tree;
- NOC: Number of Children;
- CBO: Coupling Between Object classes;

- RFC: Response for a Class;
- LCOM: Lack of cohesion in methods;
- Ca: Afferent couplings;
- NPM: Number of public methods;
- SLOC: Source Line of Code;
- CC: cumulative McCabe Cyclomatic Complexity;
- Calls: the number of methods calls into the class;
- Uses: the number of attribute accesses into the class.

Table 3 presents the descriptive statistics of the independent variables computed on the latest version of our project. The data set contains 1,516 bugs that are mapped over 2,069 faulty classes. The non-faulty classes are 2,744, in total we have 4,813 classes and not 10,716 because we consider only the latest version of the software.

Table 3. Descriptive statistics of the collected software metrics.

Indep. Var.	AVG	Std. Dev.	Min	P25	Median	P75	Max
WMC	66.80	152.95	1	12	27	67	3,415
DIT	1.78	1.15	1	1	1	2	8
NOC	0.82	4.98	0	0	0	0	147
CBO	17.08	18.08	0	7	12	21	354
RFC	70.58	87.86	1	24	46	83	14,378
LCOM	570.65	5,394.85	0	2	23	150	214,714
Ca	10.25	66.25	0	0	1	4	2,474
NPM	13.49	25.26	0	2	5	14	571
SLOC	16.28	27.96	1	4	8	18	581
CC	15.56	27.05	1	3	8	17	516
Calls	78.43	145.23	0	19	40	83	3,142
Uses	25.33	49.77	0	6	12	27	1,575

We examine the correlation between the variables using the Pearson Correlation Matrix (Table 5). After that, we keep only five independent variables, they are RFC, LCOM, Ca, CC, Uses. We choose these variables because the other ones (WMC, CBO, NPM, SLOC, Calls) are highly correlated with the first set (see bold values in Table 5). We discarded DIT and NOC because they have very low variance (see Table 3), so estimates of the model parameters were unstable.

4.2 Elaboration and result

To improve the stability of the model, we did a PCA transformation of the selected independent variables. We trained the model using 10-folds cross-validation, using only 67% of the available examples. In this subset, there are 3,231 classes (1,371 faulty, 1,860 non-faulty) that are extracted from the original data set using the stratified sampling algorithm. In this way, we can test our model on "fresh" data, this can give us a better estimation of the model performances in the real world. The contingency matrix (Table 4) is computed over the entire data set (4,813 examples).

Table 4. Decision Tree (ID3Numerical) contingency matrix.

	True Not-Faulty	True Faulty	Correctness
Pred. Not-Faulty	2364	369	86.50%
Pred. Faulty	380	1700	**81.73%**
Completeness	86.15%	**82.17%**	

Low correctness means that a high percentage of the classes being classified as fault-prone do not actually contain a fault; low completeness indicates that many faults are missed. Briand et al. [4] obtained on Java source code 68% correctness, and 73% completeness. Our model has better performances, probably because we have performed this analysis on a longer period of time and on a larger code base.

Table 5. Correlation matrix of the collected software metrics.

	WMC	DIT	NOC	CBO	RFC	LCOM	Ca	NPM	SLOC	CC	Calls	Uses
WMC	1.000											
DIT	-0.066	1.000										
NOC	0.032	-0.015	1.000									
CBO	0.572	-0.011	0.006	1.000								
RFC	**0.846**	-0.045	0.030	**0.844**	1.000							
LCOM	0.617	-0.016	0.012	0.321	0.572	1.000						
Ca	0.238	-0.008	0.153	0.105	0.212	0.197	1.000					
NPM	0.663	-0.058	0.026	0.431	**0.710**	**0.756**	0.284	1.000				
SLOC	0.656	-0.014	-0.015	0.578	**0.783**	**0.660**	0.132	0.734	1.000			
CC	0.733	-0.062	0.001	0.521	**0.666**	0.258	0.090	0.335	0.515	1.000		
Calls	**0.935**	-0.047	0.028	0.726	**0.948**	0.666	0.227	0.710	0.722	0.666	1.000	
Uses	0.709	-0.085	0.006	0.510	0.721	0.519	0.177	0.640	**0.692**	0.438	0.710	1.000

5 Conclusion and future work

In this paper, we presented a case study, from where we have collected process metrics and product metrics. We have extracted software metrics from 995 daily-versions of an industrial Java project using promPM [14] infrastructure. The file version control information and the bug tracking data are extracted using ad hoc software probes. All the data are stored into a relational data warehouse (a PostgreSQL[6] instance). The mining process of this quite large amount of semi-structured data will require specific tools for the data-extraction [3] and for the data-visualization [10]. This case study represents a good opportunity to enhance the performance of the software fault-proneness metric-based prediction models available today. Another good opportunity, which makes this case study unique, is represented by the Java 1.4-1.5 migration. The source code changes can allow us to do size estimation of other codebases after migration.

In the future, we will try to reach the same conclusion of Clarke et al. [7], and we will try to improve the performances of the model using a different data set that contains the data of a large C# project.

[6] http://www.postgresql.org/

References

1. Basili, V., Briand, L., Melo, W.L.: A validation of Object-Oriented design metrics as quality indicators. IEEE TSE 22(10), 267--271 (1996)
2. Bianco, M.: SyQL: A Tool for Analyzing the Software Development Process. In: GIIS Proceedings, To appear (2009)
3. Bianco, M., Sillitti, A., Succi, G.: SyQL: an object oriented, fuzzy, temporal query language for repositories of software artifacts. In: Companion To the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, pp. 715--716 (2008)
4. Briand, L.C., Melo, W.L., Wust, J.: Assessing the applicability of fault-proneness models across object-oriented software projects. IEEE TSE 28(7), 706--720 (2002)
5. Cartwright, M., Shepperd, M.: An empirical investigation of an object-oriented software system. IEEE TSE 26(8), 786--796 (2000)
6. Chidamber, S., Kemerer, C.: A metrics suite for object oriented design. IEEE TSE 20(6), 476--493 (1994)
7. Clarke, P. J., Babich, D., King, T. M., Golam Kibria, B. M.: Analyzing clusters of class characteristics in OO applications. The Journal of System and Software 81(12), 2269--2286 (2008)
8. van Deursen, A., Kuipers, T. Building Documentation Generators. In: Proceedings of the IEEE international Conference on Software Maintenance, pp. 40 (1999)
9. Gyimóthy, T., Ference, R., Siket, L.: Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. IEEE TSE 31(10), 897--910 (2005)
10. Jermakovics, A., Moser, R., Sillitti, A., Succi, G:. Visualizing software evolution with Lagrein. In: Companion To the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, pp. 749--750 (2008)
11. Kanmani, S., Rhymend Uthariaraj, V., Sankaranarayanan, V., Thambidurai, P.: Object-oriented software fault prediction using neural networks. Information and Software Technology 49, 483--492 (2007)
12. Moonen, L.: Generating Robust Parsers Using Island Grammars. In: Proceedings of Eighth Working Conference On Reverse Engineering (2001)
13. Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: Proceedings of the 28th international Conference on Software Engineering, pp. 452--461 (2006)
14. Scotto, M., Sillitti, A., Succi, G., Vernazza, T.: A non-invasive approach to product metrics collection. Journal of System Architecture 52(11), 668--675 (2006)
15. Subramanyam, R., Krishnan, M.S.: Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. IEEE TSE 29(4), 297--310 (2003)
16. Tomaszewski, P., Håkansson, J., Grahn, H., Lundberg, L.: Statistical models vs. expert estimation for fault prediction in modified code – an industrial case study. Journal of Systems and Software 80, 1227--1238 (2007)
17. Zhou, Y., Leung, H.: Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults. IEEE TSE 32(10), 771--789 (2006)