Zoran Budimac (Ed.)

Second Workshop on Software Quality Analysis, Monitoring, Improvement and Applications

SQAMIA 2013

Novi Sad, Serbia, September 15–17, 2013

Proceedings

Department of Mathematics and Informatics
Faculty of Sciences, University of Novi Sad, Serbia
2013

Volume Editor

Zoran Budimac
University of Novi Sad
Faculty of Sciences, Department of Mathematics and Informatics
Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia
E-mail: zjb@dmi.uns.ac.rs

ISBN: 978-86-7031-269-2

Preface

This volume contains papers presented at the Second Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA 2013). SQAMIA 2013 was held during September 15 - 17, 2013, at the Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Novi Sad, Serbia.

SQAMIA 2013 is a continuation of the successful event held in 2012. Previous workshop, the first one, was organized within the 5th Balkan Conference in Informatics (BCI 2012) in Novi Sad. In 2013 SQAMIA becomes standalone event in intention to become traditional meeting of the scientists and practitioners in the field of software quality.

The main objective of SQAMIA workshop series is to provide a forum for presentation, discussion and dissemination of scientific findings in the area of software quality, and to promote and improve interaction and cooperation between scientists and young researchers from the region and beyond.

The SQAMIA 2013 workshop consisted of regular sessions with technical contributions reviewed and selected by an international program committee, as well as of invited talks presented by leading scientists in the research areas of the workshop.

SQAMIA workshops solicited submissions dealing with four aspects of software quality: quality analysis, monitoring, improvement and applications. Position papers, papers describing the work-in-progress, tool demonstration papers, technical reports or other papers that would provoke discussion were especially welcome.

In total, 13 papers were accepted and published in this proceedings volume. All published papers were double reviewed, and some papers received the attention of more than two reviewers. We would like to use this opportunity to thank all PC members and the external reviewers for submitting careful and timely opinions on papers.

Also, we gratefully acknowledge the program co-chairs, Tihana Galinac Grbac (Croatia), Marjan Heričko (Slovenia), Zoltan Horvath (Hungary), Mirjana Ivanović (Serbia) and Hannu Jaakkola (Finland), for helping to greatly improve the quality of the workshop.

We extend special thanks to the SQAMIA 2013 Organizing Committee from the Department of Mathematics and Informatics, Faculty of Sciences, especially to its chair Gordana Rakić for her hard work, diligance and dedication to make this workshop the best it can be.

Finally, we thank our sponsors, the Provincial Secretariat for Science and Technological Development, the Serbian Ministry of Education, Science and Technological Development, and the Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, for supporting the organization of this event.

And last, but not least, we thank all the participants of SQAMIA 2013 for having made all work that went into SQAMIA 2013 worthwhile.

September 2013 *Zoran Budimac*

Workshop Organization

General Chair

Zoran Budimac *(Univ. of Novi Sad, Serbia)*

Program Chair

Zoran Budimac *(Univ. of Novi Sad, Serbia)*

Program Co-Chairs

Tihana Galinac Grbac *(Univ. of Rijeka, Croatia)*
Marjan Heričko *(Univ. of Maribor, Slovenia)*
Zoltan Horvath *(Eotvos Lorand Univ., Budapest, Hungary)*
Mirjana Ivanović *(Univ. of Novi Sad, Serbia)*
Hannu Jaakkola *(Tampere Univ. of Technology, Pori, Finland)*

Program Committee

Harri Keto *(Tampere Univ. of Technology, Pori, Finland)*
Vladimir Kurbalija *(Univ. of Novi Sad, Serbia)*
Anastas Mishev *(Univ. of Ss. Cyril and Methodius, Skopje, FYR Macedonia)*
Sanjay Misra *(Atilim Univ., Ankara, Turkey)*
Vili Podgorelec *(Univ. of Maribor, Slovenia)*
Zoltan Porkolab *(Eotvos Lorand Univ., Budapest, Hungary)*
Valentino Vranić *(Slovak Univ. of Technology, Bratislava, Slovakia)*

Additional Reviewers

Roland Király *(Eotvos Lorand Univ., Budapest, Hungary)*
Miloš Radovanović *(Univ. of Novi Sad, Serbia)*

Organizing Committee (Univ. of Novi Sad, Serbia)

Gordana Rakić, Chair
Zoran Putnik
Miloš Savić

Organizing Institution

Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Serbia

Sponsoring Institutions of SQAMIA 2013

SQAMIA 2013 was partially financially supported by:

Provincial Secretariat for Science and Technological Development,
Autonomous Province of Vojvodina, Republic of Serbia
Ministry of Education, Science and Technological Development,
Republic of Serbia
Department of Mathematics and Informatics, Faculty of Sciences,
University of Novi Sad, Serbia

Table of Contents

◦ Stability of Software Defect Prediction in Relation to Levels of Data Imbalance 1
Tihana Galinac Grbac, Goran Mauša, and Bojana Dalbelo-Bašić

◦ Enhancing Software Quality in Students' Programs 11
Stelios Xinogalos and Mirjana Ivanović

◦ Using Object Oriented Software Metrics for Mobile Application Development 17
Gregor Jošt, Jernej Huber, and Marjan Heričko

◦ Beta Testing of a Mobile Application: A Case Study 29
Mateja Kocbek and Marjan Heričko

◦ Two-dimensional Extensibility of SSQSA Framework 35
Jozef Kolek, Gordana Rakić and Miloš Savić

◦ Towards New User Interfaces Based on Gesture and Sound Identification 45
Kristijan Košič, Boštjan Arzenšek, Matej Vogrničič and Saša Kuhar

◦ Software Engineering Solutions for Improving the Regression Testing Methods in Scientific Applications Development 53
Bojana Koteska, Ljupco Pejov and Anastas Mishev

◦ The Mobile Product Line 63
Luka Pavlič

◦ Transforming Low-level Languages Using FermaT and WSL 71
Doni Pracner and Zoran Budimac

◦ Documenting Software Architectures using UML 79
Filip Prentović and Zoran Budimac

◦ Redefining Software Quality Metrics to XML Schema Needs 87
Maja Pušnik, Boštjan Šumak, Zoran Budimac and Marjan Heričko

◦ SSQSA Ontology Metrics Front-End 95
Miloš Savić, Zoran Budimac, Gordana Rakić, Mirjana Ivanović and Marjan Heričko

◦ Mobile Device and Technology Characteristics' Impact on Mobile Application Testing 103
Tina Schweighofer and Marjan Heričko

Stability of Software Defect Prediction in Relation to Levels of Data Imbalance

TIHANA GALINAC GRBAC AND GORAN MAUŠA, University of Rijeka
BOJANA DALBELO–BAŠIĆ, University of Zagreb

Software defect prediction is an important decision support activity in software quality assurance. Its goal is reducing verification costs by predicting the system modules that are more likely to contain defects, thus enabling more efficient allocation of resources in verification process. The problem is that there is no widely applicable well performing prediction method. The main reason is in the very nature of software datasets, their imbalance, complexity and properties dependent on the application domain. In this paper we suggest a research strategy for the study of the performance stability using different machine learning methods over different levels of imbalance for software defect prediction datasets. We also provide a preliminary case study on a dataset from the NASA MDP open repository using multivariate binary logistic regression and forward and backward feature selection. Results indicate that the performance becomes unstable around 80% of imbalance.

Categories and Subject Descriptors: D.2.9 [**Software Engineering**]: Management—*Software quality assurance (SQA)*

Additional Key Words and Phrases: Software Defect Prediction, Data Imbalance, Feature Selection, Stability

1. INTRODUCTION

Software defect prediction is recognized as one of the most important ways to reach software development efficiency. The majority of costs during software development is spent on software defect detection activities, but their ability to guarantee software reliability is still limited. The analyses performed by [Andersson and Runeson 2007; Fenton and Ohlsson 2000; Galinac Grbac et al. 2013], in the environment of a large scale industrial software with high focus on reliability shows that faults are distributed within the system according to the Pareto principle. They prove that the majority of faults are concentrated in just small amount of system modules, and that these modules do not compose a majority of system size. This fact implies that software defect prediction would really bring benefits if a well performing model is applied. The main motivating idea is that if we were able to predict the location of software faults within the system, then we could plan defect detection activities more efficiently. This means that we would be able to concentrate defect detection activities and resources into critical locations within the system and not on the entire system.

Numerous studies have already been performed aiming to find the best general software defect prediction model [Hall et al. 2012]. Unfortunately, a well performing solution is still absent. Data in software defect prediction are very complex, and do not follow in general any particular probability distribution that could provide a mathematical model. Data distributions are highly skewed, which is connected to the popular *data imbalance problem*, thus making standard machine learning approaches inadequate. Therefore, a significant research has recently been devoted to cope with this problem.

Author's address: T. Galinac Grbac, Faculty of Engineering, Vukovarska 58, HR–51000 Rijeka, Croatia; email: tgalinac@riteh.hr; G. Mauša, Faculty of Engineering, Vukovarska 58, HR–51000 Rijeka, Croatia; email: gmausa@riteh.hr; B. Dalbelo–Bašić, Faculty of Electrical Engineering and Computing, Unska 3, HR-10000 Zagreb, Croatia; email: bojana.dalbelo@fer.hr.

In: Z. Budimac (ed.): Proceedings of the 2nd Workshop of Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA), Novi Sad, Serbia, 15.-17.9.2013, published at http://ceur-ws.org

Several solutions are offered for the data imbalance problem. However, these solutions are not equally effective in all application domains. Moreover, there is still an open question regarding the extent to which imbalanced learning methods help with learning capabilities. This question should be answered with extensive and rigorous experimentation across all application domains, including software defect prediction, aiming to explore underlaying effects that would lead to fundamental understandings [He and Garcia 2009].

The work presented in this paper is a step in that direction. We present an research strategy that aims to explore performance stability of software defect prediction models in relation to levels of data imbalance. As an illustrative example we present an experiment taken to Stability of Software Defect Prediction in Relation to Levels of Data Imbalance our strategy. We observed how learning performance, with and without stepwise feature selection, in case of logistic regression learner, is changing over a range of imbalances in the context of software defect prediction. The findings are just indicative and are to be explored by exhausting experimenting aligned with proposed strategy.

1.1 Complexity of software defect prediction data

Software defect prediction (SDP) is concerned with early prediction of system modules (file, class, module, method, component, or something else) that are likely to have a critical number of faults (above certain threshold value, THR). In numerous studies it is identified that these modules are not so common. In fact, they are special cases, and that is why they are harder to find. **Dependent variable** in learning models is usually a binary variable with two classes labeled as 'fault–prone' (FP) and 'not–fault–prone' (NFP). The number of FP modules usually is much lower, and represents a *minority class*, than the number of NFP modules which represents a *majority class*. Datasets with significantly unequal distributions of minority over majority class are *imbalanced*. **Independent variables** used in SDP studies are numerous. In this paper we will address SDP based on the static code metrics [McCabe 1976].

In SDP datasets the level of class imbalance varies for various software application domains. We reviewed the software engineering publications dealing with software defect prediction and we noticed that the percentage of the non-fault prone modules (%NFP) in the datasets varies a lot (from 1% in medical record system [Andrews and Stringfellow 2001] to more then 94% in telecom system [Khoshgoftaar and Seliya 2004]) for various software application domains (telecom industry, aeronautics, radar systems, etc.). Since there are SDP initiatives on datasets with a whole range of imbalance percentages, we are motivated to determine the percentage at which data imbalance becomes a problem, i.e., learners become unstable.

As already mentioned above, the random variables measured in software engineering usually do not follow any distribution in general, and the applicability of classical mathematical modeling methods and techniques is limited. Hence, algorithms from the machine learning have been widely adopted. Among various learning methods used in the defect prediction approaches, this paper will explore the capabilities of **multivariate binary logistic regression (LR)**. Our ultimate goal is not to validate different learning algorithms but to explore learning performance stability over different levels of imbalance. The LR has shown very good performance in the past and is known to be a simple but robust method. In [Lessmann et al. 2008] it is the 9th best classifier among 22 examined (9/22) and at the same time it is the 2nd best statistical classifier among 7 of them (2/7). The stepwise regression classifier was the most accurate classifier (1/4) and was outperformed only in cases with many outliers in [Shepperd and Kadoda 2001]. Very good performance of logistic regression was also observed in [Kaur and Kaur 2012] (3/12 it terms of accuracy and AUC), [Banthia and Gupta 2012] (1/5 both with and without preprocessing of 5 raw NASA datasets), [Giger et al. 2011] (1/8 in terms of median AUC from 15 open source projects), [Jiang et al. 2008] (2/6 in terms of AUC and 3/6 according to Nemenyi post-hoc test), etc. However, neither of the studies has analyzed the performance of logistic regression classifier in relation to data imbalance. The study [Provost 2000] assumes that in majority of published work the performance of logistic learner would be significantly improved, if it is adequately used. We will refer to this issue in more detail in Section 3.

As in the whole software engineering field, an important problem in software defect prediction is the lack of quality industrial data, and therefore generalization ability and further propagation of research results is very limited. The problem is usually that this data are considered as confidential by the industry, or the data are not available at all for industry with low maturity. To overcome these obstacles, there are initiatives for open source repositories of datasets aligned with the goal of improving generalization of research results. However, the problem of generalization still remains, because usually the open repositories contain data from a particular type of software (e.g. NASA MDP repository, open source software repositories, etc.) and/or of questionable quality [Gray et al. 2011].

In this study we used **NASA MDP datasets** and have carefully addressed all the potential issues, i.e. removed duplicates [Gray et al. 2012]. This selection is motivated by simple comparison of results with the related work, so that our contribution can be easily incorporated to the existing knowledge base of imbalance problem in the SDP area.

1.2 Experimental approach

Our goal is to explore stability of evaluation metrics for learning SDP datasets with machine learning techniques across different levels of imbalance. Moreover, we want to evaluate potential sources of bias in study design by constructing number of experiments in which we diverse one parameter per experiment. Parameters that are subject of change are explained briefly in Sect.2.

To integrate conclusions obtained from each experiment a meta–analytic statistical analysis is proposed. These methods are suggested by number of authors as tool for generalizing the results and integrating knowledge across many studies [Brooks 1997]. We propose the following steps:

(1) ***Acquiring data.*** A sample S of independent random variables $X_1, \dots, X_n$ measuring different features of a system module, and a binary dependent variable Y measuring fault–proneness (with $Y = 1$ for FP modules and $Y = 0$ for NFP modules) is obtained from a repository (e.g. open repository, open source projects, industrial projects).

(2) ***Data preprocessing.***
   (a) *Data cleaning, noise elimination, sampling.*
   (b) *Data multiplication.* From the sample S obtained in step (1) a training set of size $2/3$ the size of S and a validation set of size $1/3$ the size of S are chosen at random k times. In this way k training samples $T_1, \dots, T_k$ and k validation samples $V_1, \dots, V_k$ are obtained. These samples are categorized into ℓ categories with respect to the data imbalance defined as the percentage of the NFP modules in T_i and calculated as: $\%NFP_{T_i} = \frac{FP_{T_i}}{FP_{T_i}+NFP_{T_i}}$.
   (c) *Feature selection.* For each training set T_i a feature selection is performed. As a result some of the random variables X_j are excluded from the model. The inclusion/exclusion frequencies of X_j for each of the categories introduced in step (2b) are recorded.

(3) ***Learning.***
   (a) *Building a learning model.* A learning model is built for each training set T_i using the learning techniques under consideration.
   (b) *Evaluating model performance.* Using the validation set V_i, the model built in step (3a) is evaluated using various evaluation metrics. Let M be the random variable measuring the value of one of these metrics.

(4) ***Statistical analysis.***

(a) *Variation analysis.* The differences between ℓ samples of a random variable M obtained from samples T_i and V_i belonging to different categories introduced in step (2b) are analyzed using statistical tests. This step is repeated for each evaluation measure used in step (3b).
(b) *Cross-dataset validation.* The whole process is repeated from step (1) for m datasets from various application domains and sources. The differences between $\ell \cdot m$ samples of a random variable M are analyzed using statistical tests and the results reveal whether general behavior exists.

To summarize, the conclusions are based on the results of statistical tests comparing the mean values of performance evaluation metrics (see Table I) across different data imbalances of a training sample. The stability of performance evaluation metrics obtained with different feature selection procedures is evaluated in the same way.

2. DATA IMBALANCE

Data imbalance has received considerable attention within the data mining community during the last decade. It becomes a central point of this research, since the problem is present in a majority of data mining application areas [Weiss 2004]. In general data imbalance degrades the learning performance. The problem arises with learning accuracy of the minority class, in which we are usually more interested. Usually, we are interested to timely predict rare events represented by the minority class, for which the probability of its occurrence is low, but its occurrence leads to significant costs.

For example, suppose that only very low number of system modules is faulty, which is the case with systems with very low tolerance on failures (e.g. medical systems, aeronautic system, telecommunications, etc.). Suppose that we did not identify faulty module with the help of a software defect prediction algorithm, and due to that have developed defect detection strategy not concentrating on that particular module. Thus, we omit to identify a fault in our defect detection activity, and this fault slips to the customer site. Failure caused by this fault at customer site would then imply significant costs contained of several items: paying penalty to customer, losing customer confidence, causing additional expenses due to corrective maintenance, additional costs in all subsequent system revisions and additional cost during system evolution. This cost would be considered as misclassification cost of wrongly classified positive class (note that positive class in the context of defect prediction algorithm is a faulty module). On the other hand, misclassification cost of wrongly classified negative class would be much lower, because it would involve just more defect detection activities. Obviously, the misclassification costs are unequally weighted and this is the main obstacle in applying standard machine learning algorithms, because they usually assumes the same or similar conditions in learning and application environment [Provost 2000].

The study [Provost 2000] makes a survey of data imbalance problems and methods addressing these problems. Although different methods are recommended for data imbalance problems, it does not give definite answers regarding their applicability in the application context. Some answers are obtained by other researchers in that field afterwards, and a more recent survey is given in [He and Garcia 2009]. Still no definite guideline exists that could guide practitioners.

2.1 Dataset considerations

The most popular approach to the class imbalance problem is the usage of artificially obtained balanced dataset. There are several sampling methods proposed for that purpose. In a recent work [Wang and Yao 2013] an experiment with some of the sampling methods is conducted. However, it is concluded in [Kamei et al. 2007] that sampling did not succeed to improve performance with all the classifiers. In [Hulse et al. 2007] it is identified that classifier performance is improved with sampling, but individual learners respond differently on sampling.

Another problem with datasets is that in practice, the datasets are often very complex, involving a number of issues like overlapping, lack of representative data, within and between class imbalance, and often high dimensionality. The effects of these issues were widely analyzed separately sample size in [Raudys and Jain 1991], dimensionality reduction: [Liu and Yu 2005], noise elimination [Khoshgoftaar et al. 2005], but not in conjunction with the data imbalance. The study performed in [Batista et al. 2004] observes that the problem is related to a combination of absolute imbalance and other complicating factors. Thus, the imbalance problem is just an additional issue in complex datasets such as datasets for software defect prediction.

Different aspects of feature selection in relation to class imbalance has been studied in [Khoshgoftaar et al. 2010; Gao and Khoshgoftaar 2011; Wang et al. 2012]. All these studies were performed on datasets from the NASA MDP repository. In this work we also used a stepwise feature selection as a preprocessing step, because the dataset is high dimensional and we experiment with logistic regression. Hence, we were able to investigate the stability of the performance with and without feature selection procedure over different levels of imbalance.

Besides the methods explained above for obtaining artificially balanced datasets, another approach is to adapt standard machine learning algorithms to operate for imbalance datasets. In that case the learning approach should be adjusted to the imbalanced situation. A complete review of such approaches and methods can be found in [He and Garcia 2009].

2.2 Evaluation metrics

Another problem of standard machine learning algorithms for imbalanced data is in usage of inadequate evaluation metrics during learning procedure or to evaluate final result. Evaluation metrics are usually derived from the confusion matrix and are given in Table I. They are defined in terms of the following score values. A true positive (TP) score is counted for every correctly (true) classified fault-prone module, and a true negative (TN) score for every correctly (true) classified non-fault-prone module. The other two possibilities are related to false prediction. A false positive (FP) score is counted for every false classified or misclassified non-fault-prone module (often referred to as Type II error), and a false negative (FN) score is counted for every false classified or misclassified fault-prone module (often referred to as Type I error) [Runeson et al. 2001; Khoshgoftaar and Seliya 2004]. For example, classification accuracy ACC, the most commonly used evaluation metric in standard machine learning algorithms, is not able to value the minority class appropriately, and leads to poor classification performance of minority class.

In the case of class imbalance, the precision (PR) and recall (TPR) metrics given in Table I are recommended in number of studies [He and Garcia 2009], as well as the F–measure and G–mean which are not used here. The precision and recall in combination give a measure of correctly classified fault–prone modules. Precision measures exactness, i.e., how many fault–prone modules are classified correctly, and recall measures completeness, i.e., how many fault–prone are classified correctly.

Table I. Evaluation metrics

Metrics	Definition	Formula
Accuracy (ACC)	number of correctly classified modules divided by total modules	$\frac{TP+TN}{TP+FP+TN+FN}$
True positive rate (TPR) (sensitivity, recall)	number of correctly classified fault–prone modules divided by total number of fault-prone modules	$\frac{TP}{TP+FN}$
Precision (PR) (positive predicted value)	number of correctly classified fault–free modules divided by total number of fault–free modules	$\frac{TP}{TP+FP}$

The output of a probabilistic machine learning classifier is the probability for a module to be fault-prone. Therefore, a cutoff percentage has to be defined in order to perform classification. Since choosing a cutoff value leaves room to bias and possible inconsistencies in a study [Lessmann et al. 2008], there is another measure that deals with that problem called the area under curve, AUC [Fawcett 2006]. It takes into account the dependence of TPR and a similar metric for false positive proportion on the cutoff value.

All of the aforementioned techniques are not cost sensitive, and in the case of rare cases with very high misclassification cost of type I error the key performance indicator is cost. The most favorable evaluation criteria for imbalanced datasets are cost curves and is also recommended in [Jiang et al. 2008] for SDP domain.

3. PRELIMINARY CASE STUDY

To illustrate the application of the research strategy proposed in Section 1.2, verify strategy, provide evidence for the dependence of the machine learning performance on the level of data imbalance, and indicate our future goals, we have undertaken a preliminary case study.

(1) Dataset KC1 from NASA MDP repository has been acquired. It consists of $n = 29$ features, i.e., independent variables X_j. The dependent variable in this dataset is the number of faults in a system module. From this variable we derived binary dependent variable Y by setting ten different thresholds for fault proneness, from 1 to 19 with step of 2 (1, 3, 5,...). In this way we obtained ten different samples S and we continue the analysis for all of them.

(2) (a) The well known issues with the dataset are eliminated using data cleaning tool [Shepperd et al. 2013].

(b) For each of the ten samples obtained in step (1), we made 50 iterations of the random splitting into training and validation samples. Thus we obtained $k = 500$ samples T_i and V_i with the range of data imbalance from 51% to 96%. The samples are categorized into $\ell = 5$ categories of equal length (each spanning 9%).

(c) In the case study we also consider the influence of a feature selection procedure, as already mentioned in 2. We consider the forward and backward stepwise selection procedure [Han and Kambar 2006]. The decision for inclusion and exclusion of a feature is based on level of statistical significance, the $p - value$. The common significance levels for inclusion and exclusion of features are used as in [Mausa et al. 2012; Briand et al. 2000] with $p\_in = 0.05$ and $p\_out = 0.1$ respectively. The percentage of inclusion of a feature for both procedures and different categories of data imbalance are given in Table II. We conclude that feature selection stability of some features is very tolerant to data imbalance (e.g. Feature 5, 22, 28, 29 is always excluded, for both forward and backward model). Some features are very stable until certain level of balance (for example Feature 2 is always included 100% until category with data imbalance of 78%). It is also interesting to observe that some features have similar feature selection stability in ideal balance case and highly imbalanced case, whereas for moderate imbalance have opposite feature selection decision.

(3) (a) Learning models are built using multivariate binary logistic regression (LR) [Hastie et al. 2009]. The model incorporates more than just one predicting variable and in fault predicting case performs according to the equation

$$\pi\left(X_1, X_2, ...X_n\right) = \frac{e^{C_0+C_1X_1+...+C_nX_n}}{1+e^{C_0+C_1X_1+...+C_nX_n}}, \tag{1}$$

where C_j are the regression coefficients corresponding to X_j, and π is the probability that a fault was found in a class during validation. In order to obtain a binary outgoing variable,

Table II. Percentage of inclusion of a feature

	'51% -60%'		**'60% -69%'**		**'69% -78%'**		**'78% -87%'**		**'87% -96%'**	
	Forw.	Back.	Forw.	Back.	Forw.	Back.	Forw.	Back.	Forw.	Back.
Ft. 1	4,8%	26,2%	1,7%	20,3%	0,0%	9,1%	5,2%	12,2%	14,8%	15,8%
Ft. 2	100,0%	100,0%	100,0%	100,0%	95,5%	100,0%	73,3%	72,7%	44,3%	43,2%
Ft. 3	78,6%	76,2%	91,5%	84,7%	34,1%	77,3%	30,8%	63,4%	16,9%	38,3%
Ft. 4	2,4%	9,5%	10,2%	45,8%	18,2%	52,3%	0,6%	6,4%	0,0%	1,6%
Ft. 5	7,1%	14,3%	5,1%	16,9%	2,3%	22,7%	1,2%	27,3%	2,2%	7,1%
Ft. 6	0,0%	11,9%	5,1%	11,9%	0,0%	4,5%	2,3%	16,9%	3,8%	29,0%
Ft. 7	9,5%	23,8%	42,4%	47,5%	15,9%	50,0%	4,7%	13,4%	0,0%	9,8%
Ft. 8	2,4%	14,3%	8,5%	16,9%	15,9%	20,5%	55,2%	52,9%	84,2%	88,0%
Ft. 9	4,8%	28,6%	10,2%	37,3%	0,0%	22,7%	3,5%	26,7%	0,5%	19,1%
Ft. 10	2,4%	23,8%	6,8%	32,2%	2,3%	20,5%	1,2%	37,2%	2,2%	60,1%
Ft. 11	0,0%	11,9%	0,0%	15,3%	2,3%	43,2%	20,9%	40,7%	13,7%	17,5%
Ft. 12	7,1%	23,8%	1,7%	22,0%	54,5%	86,4%	35,5%	65,1%	9,8%	26,8%
Ft. 13	9,5%	42,9%	20,3%	39,0%	0,0%	27,3%	5,8%	43,6%	2,2%	60,1%
Ft. 14	21,4%	19,0%	16,9%	16,9%	0,0%	6,8%	2,9%	15,1%	0,0%	13,1%
Ft. 15	4,8%	23,8%	5,1%	13,6%	0,0%	13,6%	0,6%	8,1%	6,0%	19,1%
Ft. 16	11,9%	21,4%	1,7%	6,8%	0,0%	25,0%	1,2%	15,7%	2,2%	19,1%
Ft. 17	4,8%	40,5%	1,7%	27,1%	0,0%	11,4%	4,7%	14,0%	17,5%	23,5%
Ft. 18	9,5%	16,7%	27,1%	18,6%	2,3%	38,6%	18,6%	26,2%	12,6%	21,3%
Ft. 19	7,1%	11,9%	25,4%	37,3%	0,0%	9,1%	1,2%	27,3%	0,0%	21,9%
Ft. 20	0,0%	45,2%	1,7%	54,2%	0,0%	45,5%	0,0%	43,6%	2,2%	22,4%
Ft. 21	4,8%	16,7%	0,0%	39,0%	0,0%	27,3%	0,0%	31,4%	0,0%	44,3%
Ft. 22	0,0%	9,5%	3,4%	8,5%	0,0%	0,0%	0,0%	2,3%	0,0%	0,5%
Ft. 23	7,1%	21,4%	16,9%	30,5%	0,0%	6,8%	1,2%	26,2%	0,0%	13,1%
Ft. 24	0,0%	21,4%	0,0%	27,1%	0,0%	20,5%	0,0%	20,9%	0,5%	20,2%
Ft. 25	11,9%	35,7%	10,2%	71,2%	0,0%	9,1%	1,7%	18,0%	10,9%	29,5%
Ft. 26	7,1%	52,4%	1,7%	59,3%	0,0%	20,5%	0,0%	30,8%	1,1%	33,3%
Ft. 27	35,7%	50,0%	8,5%	20,3%	0,0%	4,5%	2,3%	16,9%	11,5%	23,0%
Ft. 28	2,4%	14,3%	11,9%	13,6%	6,8%	4,5%	0,6%	10,5%	1,6%	15,3%
Ft. 29	0,0%	11,9%	1,7%	3,4%	4,5%	0,0%	0,6%	2,3%	0,0%	1,6%

a *cutoff value* splits the results into two categories. Researchers often set the cutoff value to 0.5 [Zimmermann and Nagappan 2008]. However, the logistic regression is also robust to data imbalance and this robustness is achieved with setting of cutoff value to optimal value dependent on misclassification costs [Basili et al. 1996]. Our goal is to explore learning performance over different imbalance levels. However, in this study, due to space limitation, we provide preliminary results exploring learning performance stability of standard learning algorithms. Therefore, we provide results of experiments with cutoff value set to 0.5 (that is how standard learning algorithms equally weight misclassification costs). We considered there three different models (with forward feature selection, backward feature selection and without feature selection) and for each of these models, the coefficients are calculated separately.

(b) For all validation samples from step (2b) we count the TN, TP, FN and FP scores of the corresponding model, and calculate the learning performance evaluation metrics ACC, TPR (Recall), AUC and Precision using formulas in Table I.

(4) We made a statistical analysis of the behavior of evaluation metrics measured in step (3b) between different categories introduced in step (2c). Since the samples are not normally distributed, we used the non-parametric tests. The Kruskal-Wallis test showed for all metrics that the values depend on the category. To explore the differences further, we applied multiple comparison test. It reveals that all considered evaluation metric become unstable at the level of imbalance of 80%. According to the theory explained in section 2, we expect that we will get significantly different mean values for all metrics in category of highest data imbalance (90% - 100%).

4. DICUSSION

Data imbalance problem has been widely investigated and there were numerous approaches studying its effects aiming to propose a general solution to that problem. However, from the experiments in machine learning theory it becomes obvious that this is not only related to proportion of minority over majority class but there are also other influences present in complex datasets. As the datasets in software defect prediction (SDP) research area are usually extremely complex, there is a huge unexplored area of research related to applicability of these techniques in relation to the level of data imbalance. That is exactly our main motivation for this work.

There are many approaches, depending on particular dataset, to SDP and development of the learning model. Since we are interested in the performance stability of machine learners over SDP datasets, we should rigorously explore the strengths and limitations of these approaches in relation to the level of data imbalance. Therefore, we present an exploratory research strategy and an example of a case study performed according to this strategy. Although, we use our experiment to eliminate as much as possible inconsistencies and threats of applying the strategy, there is still place for improvement.

In our case study we present how performance stability is significantly degraded at a higher level of imbalance. This confirms the results obtained by other researchers using different approaches. That conclusion have proved reliability of our strategy. Moreover, with the help of our research strategy we confirmed that feature selection becomes instable with higher data imbalance. We have also observed that the feature selection is consistent across levels of imbalance for some features.

Future work should involve extensive exploration of SDP datasets with the proposed strategy. Our vision is that at the end we can gain deeper knowledge about imbalanced data in SDP and applicability of techniques in different levels of imbalance. Finally, we would like to categorize datasets using the proposed strategy and results of this exhaustive research that would serve as a guideline for practitioners while developing software defect prediction model.

REFERENCES

C. Andersson and P. Runeson. A replicated quantitative analysis of fault distributions in complex software systems. *IEEE Trans. Softw. Eng.*, 33(5):273–286, May 2007.

A. Andrews and C. Stringfellow. Quantitative analysis of development defects to guide testing: A case study. *Software Quality Control*, 9:195–214, November 2001.

D. Banthia and A. Gupta. Investigating fault prediction capabilities of five prediction models for software quality. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1259–1261, New York, NY, USA, 2012. ACM.

V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Engineering*, 22(10):751–761, October 1996.

G. E. A. P. A. Batista, R. C. Prati, and M.C. Monard. A study of the behavior of several methods for balancing machine learning training data. *SIGKDD Explor. Newsl.*, 6(1):20–29, 2004.

L. C. Briand, J. W. Daly, V. Porter, and J. Wüst. A comprehensive empirical validation of product measures for object-oriented systems, 1998.

L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter. Exploring the relationship between design measures and software quality in object-oriented systems. *J. Syst. Softw.*, 51:245–273, May 2000.

A. Brooks. Meta Analysis–A Silver Bullet for Meta-Analysts. *Empirical Softw. Engg.*, 2(4):333–338, 1997.

T. Fawcett. An introduction to ROC analysis. *Pattern Recogn. Lett.*, 27(8):861–874, Aug. 2006.

N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Softw. Eng.*, 26(8):797–814, Aug. 2000.

K. Gao and T. M. Khoshgoftaar. Software defect prediction for high-dimensional and class-imbalanced data. In *SEKE*, pages 89–94. Knowledge Systems Institute Graduate School, 2011.

E. Giger, M. Pinzger, and H. C. Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 83–92, New York, NY, USA, 2011. ACM.

D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson. The misuse of the nasa metrics data program data sets for automated software defect prediction. *Processing*, pages 96–103, 2011.

D. Gray, D. Bowes, N. Davey, Y. Sun and B. Christianson. Reflections on the NASA MDP data sets. *IET Software*, pages 549 5583, 2012.

T. Galinac Grbac, P. Runeson, and D. Huljenic. A second replicated quantitative analysis of fault distributions in complex software systems. *IEEE Transactions on Software Engineering*, 39(4):462–476, 2013.

T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, 38(6):1276–1304, 2012.

J. Han and M. Kamber. *Data mining: concepts and techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition, 2009.

H. He and E. A. Garcia. Learning from Imbalanced Data. *IEEE Trans. Knowledge and Data Engineering*, 21(9):1263-1284, 2009.

J. Hulse, T. Khoshgoftaar, A. Napolitano. Experimental perspectives on learning from imbalanced data. In *in Proc. 24th international conference on Machine learning (ICML '07)*, pages 935–942. 2007.

Y. Jiang, B. Cukic, and Y. Ma. Techniques for evaluating fault prediction models. *Empirical Softw. Engg.*, 13:561–595, October 2008.

Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, K. Matsumoto. The Effects of Over and Under Sampling on Fault-prone Module Detection. In *in Proc. ESEM 2007, First International Symposium on Empirical Software Engineering and Measurement*, pages 196–201. IEEE Computer Society Press, 2007.

I. Kaur and A. Kaur. Empirical study of software quality estimation. In *Proceedings of the Second International Conference on Computational Science, Engineering and Information Technology*, CCSEIT '12, pages 694–700, New York, NY, USA, 2012. ACM.

T. M. Khoshgoftaar, E. B. Allen, R. Halstead, and G. P. Trio. Detection of fault-prone software modules during a spiral life cycle. In *Proceedings of the 1996 International Conference on Software Maintenance*, ICSM '96, pages 69–76, Washington, DC, USA, 1996. IEEE Computer Society.

T. M. Khoshgoftaar and N. Seliya. Comparative assessment of software quality classification techniques: An empirical case study. *Empirical Softw. Engg.*, 9(3):229–257, Sept. 2004.

T. M. Khoshgoftaar, N. Seliya, K. Gao. Detecting noisy instances with the rule-based classification model. *Intell. Data Anal.*, 9(4):347–364, 2005.

T. M. Khoshgoftaar, K. Gao, N. Seliya. Attribute Selection and Imbalanced Data: Problems in Software Defect Prediction *In Proceedings: the 22nd IEEE International Conference on Tools with Artificial Intelligence*, 137-144, 2010.

H. Liu, L. Yu. Toward Integrating Feature Selection Algorithms for Classification and Clustering. *IEEE Trans. on Knowl. and Data Eng.*, 17(4):491–502, 2005.

S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: a proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.

G. Mausa, T. Galinac Grbac, and B. Basic. Multivariate logistic regression prediction of fault-proneness in software modules. In *MIPRO, 2012 Proceedings of the 35th International Convention*, pages 698–703, 2012.

T.J. McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

N. Ohlsson, M. Zhao, and M. Helander. Application of multivariate analysis for software fault prediction. *Software Quality Control*, 7:51–66, May 1998.

F. Provost. Machine Learning from Imbalanced Data Sets 101. In *Proc. Learning from Imbalanced Data Sets: Papers from the Am. Assoc. for Artificial Intelligence Workshop*, Technical Report WS-00-05, 2000.

S. J. Raudys, A. K. Jain. Small Sample Size Effects in Statistical Pattern Recognition: Recommendations for Practitioners. *IEEE Trans. Pattern Anal. Mach. Intell.*, 13(3):252–264, May 1991.

P. Runeson, M. C. Ohlsson, and C. Wohlin. A classification scheme for studies on fault-prone components. In *Proceedings of the Third International Conference on Product Focused Software Process Improvement*, PROFES '01, pages 341–355, London, UK, 2001. Springer-Verlag.

M. Shepperd and G. Kadoda. Comparing software prediction techniques using simulation. *IEEE Trans. Softw. Eng.*, 27(11):1014–1022, Nov. 2001.

M. Shepperd, Q. Song, Z. Sun, C. Mair Data Quality: Some Comments on the NASA Software Defect Data Sets. *IEEE Trans. Softw. Eng.*, http://doi.ieeecomputersociety.org/10.1109/TSE.2013.11, Nov. 2013.

H. Wang, T. M. Khoshgoftaar, and A. Napolitano. An Empirical Study on the Stability of Feature Selection for Imbalanced Software Engineering Data. *In Proceedings of the 2012 11th International Conference on Machine Learning and Applications - Volume 01*, ICMLA '12, pages 317–323, Washington, DC, USA, 317–323.

S. Wang and X. Yao. Using Class Imbalance Learning for Software Defect Prediction. *IEEE Transactions on Reliability*, 62(2):434 - 443, 2012.

G.M. Weiss. Mining with rarity: a unifying framework. In *SIGKDD Explor. Newsl.*, 6(1):7–19, 2004.

T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 531–540, New York, NY, USA, 2008. ACM.

Enhancing Software Quality in Students' Programs

STELIOS XINOGALOS, University of Macedonia
MIRJANA IVANOVIĆ, University of Novi Sad

This paper focuses on enhancing software quality in students' programs. To this end, related work is reviewed and proposals for applying pedagogical software metrics in programming courses are presented. Specifically, we present the main advantages and disadvantages of using pedagogical software metrics, as well as some proposals for utilizing features already built in contemporary programming environments for familiarizing students with various software quality issues. Initial experiences on usage of software metrics in teaching programming courses and concluding remarks are also presented.

Categories and Subject Descriptors: D.2.8 [**Software Engineering**]: Metrics – *Complexity measures*; K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer science education*

General Terms: Education, Measurement

Additional Key Words and Phrases: Pedagogical Software Metrics, Quality of Students' Software Solutions, Assessments of Students' Programs

1. INTRODUCTION

Teaching and learning programming presents teachers and students respectively with several challenges. Students have to comprehend the basic algorithmic/programming constructs and concepts, acquire problem solving skills, learn the syntax of at least one programming language, and familiarize with the programming environment and the whole program development process. Moreover, students nowadays have to familiarize with the imperative and object-oriented programming techniques and utilize them appropriately. The numerous difficulties encountered by students regarding these issues have been recorded in the extended relevant literature. Considering time restrictions, large classes and increasing dropout rates the chances to add more important software development aspects in introductory programming courses, such as software quality aspects, seems to be a difficult mission.

On the other hand, several empirical studies regarding the development of real-world software systems have shown that 40% to 60% of the development resources are spent on testing, debugging and maintenance issues. It is clear both for the software industry and those teaching programming that the students should be educated to write code of better quality. Several efforts have been made from researchers and teachers towards achieving this goal. These efforts focus mainly on:

– adjusting widely accepted software quality metrics for use in a pedagogical context,
– devising special tools that carry out static code analysis of students' programs.

This paper focuses on studying the related work and making some proposals for dealing with software quality in students' programs. Specifically, we propose utilizing features already built in contemporary programming environments used in our courses, for presenting and familiarizing students with various software quality issues without extra cost. Of course, using pedagogical software metrics is not an issue that refers solely to pure programming courses. However, since students formulate their programming style in the context of introductory programming courses, it is important to introduce pedagogical software metrics in such courses and then extend on other software engineering, information systems and database courses, or generally in courses that require from students to develop software. The rest of the

This work was partially supported by the Serbian Ministry of Education, Science and Technological Development through project *Intelligent Techniques and Their Integration into Wide-Spectrum Decision Support*, no. OI174023 and by the Provincial Secretariat for Science and Technological Development through multilateral project "*Technology Enhanced Learning and Software Engineering*".
Authors' addresses: S. Xinogalos, Department of Applied Informatics, School of Information Sciences, 156 Egnatia str., 54006 Thessaloniki, Greece, email: stelios@uom.gr; M. Ivanović, Department of Mathematics and Informatics, Faculty of Sciences, Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia, email: mira@dmi.uns.ac.rs

In: Z. Budimac (ed.): Proceedings of the 2nd Workshop of Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA), Novi Sad, Serbia, 15.-17.9.2013, published at http://ceur-ws.org

paper is organized as follows. In the 2nd section we refer to adequate related work. Section 3 considers usage of pedagogical software metrics. In section 4 we present some initial experiences of usage of software metrics in teaching programming courses. Last section brings concluding remarks.

2. RELATED WORK

When we refer to commercial software quality a long list of software metrics exists that includes basic metrics and more elaborated ones, as well as combinations and variations of them. Highly referenced basic metrics are: (i) the *Healstead metric* that is used mainly for estimating the programming effort of a software system in terms of the operators and operands used; and (ii) the *McCabe cyclomatic complexity measure* that analyzes the number of the different execution paths in the system in order to decide how complex, modular and maintainable it is.

The problem with such metrics is that they have not been developed for use in a pedagogical context. As [Patton and McGill 2006] state such metrics have the potential to be utilized for analysis of students' programs, but they have specific shortcomings: several metrics give emphasis on the length of the code irrespectively of its logic and do not differentiate between various uses of language features, such as a *for* versus a *while* loop, or a *switch-case* versus a sequence of *if* statements. When we talk about students' programs it is clear that as educators we consider the logic of a program more important than its length, while the appropriate utilization of language features is one of the main goals of introductory programming courses. In this sense, researchers have proposed software metrics specifically for analyzing student produced software.

One such framework has been proposed by [Patton and McGill, 2006] and includes the following elements: [1] *language vocabulary*: use of targeted language constructs and elements (e1); [2] *orthagonality/encapsulation*: of both tasks (e2) and data (e3); [3] *decomposition/modularization*: avoiding duplicates of code (e4) and overly nested constructs (e5); [4] *indirection and abstraction* (e6); [5] *polymorphism, inheritance and operator overloading* (e7).

Patton and McGill [2006] devised this framework in the context of a study regarding optimal use of students' software portfolios and propose attributing its elements to specific pedagogical objectives, and weighting them according to the desired outcomes of the institution and instructor.

Another recent study aimed at devising a list of metrics for measuring static quality of student code and at the same time utilizing it for measuring quality of code between first and second year students. In this study, seven code characteristics (in italics) that should be present in students' code are analyzed in 22 properties, as follows [Breuker et al. 2011]: [1] *size-balanced*: (p1) number of classes in a package; (p2) number of methods in a class; (p3) number of lines of code in a class; (p4) number of lines of code in a method, [2] *readable*: (p5) percentage of blank lines; (p6) percentage of (too) long lines, [3] *understandable*: (p7) percentage of comment lines; (p8) usage of multiple languages in identifier naming; (p9) percentage of short identifiers, [4] *structure*: (p10) maximum depth of inheritance; (p11) percentage of static variables; (p12) percentage of static methods; (p13) percentage of non-private attributes in a class, [5] *complexity*: (p14) maximum cyclomatic complexity at method level; (p15) maximum level of statement nesting at method level, [6] *code duplicates*: (p16) number of code duplicates; (p17) maximum size of code duplicates, [7] *ill-formed statements*: (p18) number of assignments in an 'if' or 'while' condition; (p19) number of 'switch' statements without 'default'; (p20) number of 'breaks' outside a 'switch' statement; (p21) number of methods with multiple 'returns'; (p22) number of hard-coded constants in expressions.

Some researchers have moved a step forward and have developed special tools that perform static analysis of students' code. Two characteristic examples are CAP [Schorsch 1995] and Expresso [Hristova et al. 2003]. *CAP* ("Code Analyzer for Pascal") analyzes programs that use a subset of Pascal and provides user-friendly and informative feedback for syntax, logic and style errors, while Expresso aims to assist novices writing Java programs in fixing syntax, semantic and logic errors, as well as contributing in acquiring better programming skills.

Several other tools have been developed with the aim of automatic grading of students' programs in order to provide them with immediate feedback, reducing the workload for instructors and also detecting plagiarism [Pribela et al. 2008, Truong et al. 2004]. However, in most cases these environments are targeted to specific languages, such as CAP for Pascal and Expresso for Java. A platform and programming language independent approach is presented in [Pribela et al. 2012]. Specifically, the usage of software metrics in automated assessment is studied using two language independent tools: SMILE for calculating software metrics and Testovid for automated assessment.

However, none of these solutions has gained widespread acceptance. Our proposal is to utilize features of contemporary programming environments and tools in order to teach and familiarize students with important aspects of software quality, as well as help them acquire better programming habits and skills without extra cost. Usually features of this kind are not utilized appropriately, although they provide the chance to help students increase the quality of their programs easily.

3. USING PEDAGOGICAL SOFTWARE METRICS

3.1 Advantages and Disadvantages

Pedagogical software metrics can be applied with various ways in courses having a software aspect with the ultimate goal of developing better quality software. Specifically, they can be given to students just as guidelines to follow in order to develop quality code, or as factors that count towards grading their software products. In the latter case it is clear that a considerable amount of time should be devoted in training students in comprehending and applying the selected software metrics. On the one hand this is important to take place even from introductory programming courses, since this is the time when students formulate their "good" or "bad" programming style/habits that is not easy to change in the future. On the other hand, novices have several difficulties to deal with when introduced to programming and adding formal rules regarding software metrics might not be a good choice at least not for all students. Moreover, adding more material in introductory programming courses is not easy in terms of both time and volume of material.

Several researchers and instructors have integrated software metrics in systems used for automatic checking of software developed by students, either for grading their programs or/and for detecting plagiarism. The advantages are several. First of all, students can get immediate feedback about their achievements and be supported in overcoming their difficulties and misconceptions, while grading is fair. Secondly, instructors save a great deal of time from correcting programs, a process that in the case of large classes and many practical exercises is extremely time-consuming. Of course, developing such tools is also not easy and requires a great deal of time and effort.

3.2 The Educational IDE BlueJ

The educational programming environment BlueJ is a widely known environment used in introductory object oriented programming courses, since it offers several pedagogical features that assist novices. These features can be appropriately utilized for teaching and familiarizing students with software quality aspects described in the previous section and helping them acquire better programming habits.

Editor features. The editor of BlueJ provides some features that can help students firstly appreciate a good style of programming and secondly inspect their code for the existence of properties proposed in the framework by [Breuker et al., 2011] or the elements proposed by [Patton and McGill, 2006], or other similar frameworks. These features are:

– *line numbers* that can be used for a quick look on the lines of code in methods (p4) and classes (p3) if the instructor considers it important and provides students with relevant measures for a project
– *auto-indenting* and *bracket matching* help students write code that is better structured and more readable. However, several times students do not consider style so important and they write endless lines of code with no indentation and no distinction between blocks of code. In the case of errors that are so common in student's programs, this lack of structure makes the detection of errors difficult especially in the case of nested constructs (e5). The instructor can easily convey this concept to students by presenting students such a program (or using their own ones) and

using the *automatic-layout* ability provided for BlueJ for presenting them the corresponding program with proper indentation in order to help them realize the difference in practice.

- *syntax-highlighting* can help students easily inspect their code for ill-formed statements (p18-p21). However, the instructor has to make students comprehend that they have to inspect the code they write and not just compile and run it. Syntax-highlighting, for example, can help students easily detect a sequence of 'if' statements that should be replaced by an "if/else if..' or 'switch' construct.
- *scope highlighting* that is presented with the use of different background colors for blocks of code should be – in the same sense as above – utilized for a quick inspection of nested constructs (e5) and level of statement nesting at method level (p15) in order to avoid increased complexity. The instructor can give students some maximum values to have in mind and ring them a bell for reconsidering the decomposition/modularization of their solution.
- *method comments* can be easily added in students' code. When the cursor is in the context of a method and the student invokes the 'method comment' choice a template of a method comment is added in the source code containing the method's name, java doc tags and basic information regarding parameters, return types and so on. Students must understand that comments (p7) produce more readable and maintainable code and also can be used for producing a more comprehendible and valuable *documentation view* of class. This interface of a class is important in project teams and the development of real world software systems.

Moreover, if instructors think that a more formal approach should be adopted towards checking coding styles the BlueJ *CheckStyle* extension [Giles and Edwards] can be used. This extension is a wrapper for the CheckStyle (release v5.4) development tool and allows the specification of coding styles in an external file.

Incremental development and testing. Students tend to write large portions of code before they compile and test it, increasing this way the possibility for error-prone code of less quality. We consider that it is important to develop and test a program incrementally in order to achieve better quality code. BlueJ offers some unique possibilities for novices towards this direction. Specifically, the ability of *creating objects and calling methods with direct manipulation techniques* makes incremental development and testing an easy process. Students are encouraged to create instances of a class (by right-clicking on it from the simplified UML class diagram of a project presented in the main window of BlueJ) and call each method they implement for testing its correctness. Students can even call a method by passing it - with direct manipulation techniques - references to objects existing and depicted in the *object bench*. This makes incremental developing and testing of each method much easier and less time consuming. The invocation of methods should always be done with the *object inspector* of each object active, in order to check how the object's state is affected and also how it affects method execution. Students should be encouraged to use the *object inspector* to check: encapsulation of data (e3); static variables (p11); private and non-private attributes (p13). It is not unusual for students to write code mechanically and so it is important for them to learn to inspect afterwards what they have written. This also stands out for methods as well. The pop-up menu with the available methods for an object of a class, shows explicitly the public interface of a class and can help novices comprehend public and private access modifiers in practice and utilize them appropriately. Also, the dialog box that appears when a student creates an object or calls a method for an object, "asks" the student to enter a value of the appropriate type for each parameter and helps students realize whether their choices of parameters were correct (i.e. a parameter is missing or it is not needed). Students can experiment with all the aforementioned concepts by writing the corresponding statements in the *Code pad* incorporated in the main window of BlueJ.

Visualization of concepts. The main window of BlueJ presents students with a simplified *UML class diagram* giving an overview of a project's *structure*. Specifically, the following information is presented: name of each class; type of class (concrete, abstract, interface, applet, enum); 'uses' and 'inheritance' relation. This UML class diagram can be used for getting an overview of a project either it is given to students for studying it or it is developed by students themselves. Students can easily inspect the overall structure of a project, the number of classes (p1) and the depth of inheritance (p10). Students should also be encouraged to inspect the UML class diagram in order to: detect classes representing related entities that have been defined independently and in this case refactor the project using inheritance; check for cohesion and coupling and validate the decisions made while coding. In this process, the *Class card extension* [Steinhuber, 2008] can support further students, since it can present (in a different card) for each class information regarding its attributes and methods.

Debugger. Finally, BlueJ offers a debugger that allows students to set a *breakpoint* in the desired source code line and execute the code in a *step by step* manner, watching the *call sequence stack*, inspecting the values of *static*, *instance* and *local variables*.

4. SOME EXPERIENCES ON USING PEDAGOGICAL SOFTWARE METRICS

At the Department of Technology Management at the University of Macedonia in Greece pedagogical software metrics were applied, until recently that the Department was merged with the Department of Applied Informatics, only in an informal way. The Department offered a 2nd semester "Introduction to Programming" course using C as the programming language and a 3rd semester "Object Oriented Design and Programming" course based on Java. At both courses students were presented with numerous examples of good style programs and were encouraged to use them as templates for developing their programs. In the introductory course emphasis was given on the common code style conventions, ill-formed statements, level of nesting and the merits of abstraction (mainly usage of functions). At the Object Oriented Programming course that students develop programs of considerable bigger size and complexity, emphasis was given – in addition – on good structure, encapsulation, code duplicates, coupling and cohesion. However, all these were part of presenting the various programming concepts to students during lectures and practicing with them in labs, and were not presented as software metrics per se. In labs the various features of BlueJ were utilized as described in the previous section in order to help students realize the importance and adopt a good style of programming, an incremental development and testing programming strategy, as well as the habit to always inspect the structure of their programs and test its correctness. Although there is no formal assessment, the author's experience on teaching the courses the last 8 years and grading students' programs developed as homework or during exams, has showed that the majority of students apply a good style of programming, fewer students adopt an incremental development and testing strategy and even fewer test them rigorously.

At the Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad students, within "Introduction to programming" course in 1st semester using Modula-2 as the programming language, were also presented with wide range of simple examples of good style programs. Students were, during theoretical and lab exercises, encouraged using them as templates for developing their good style programs. Additionally one topic within the course is devoted to essential elements of good-style programming and also emphasis was given on the common code style conventions, ill-formed statements, level of nesting and the merits of abstraction. Later in other programming courses, especially within 3rd semester "Object Oriented Programming" course based on Java, teachers insist that students use these good style programming elements intensively in their solutions.

For lab exercises within different programming courses at our Department teachers use specially developed framework Svetovid for submission and assessment of students' software solutions. Svetovid represents a good educational tool that can be further gradually developed and enhanced. Standard part of the framework is Testovid component devoted to automatic assessment of students programming solutions. Testovid allows students to test their assignments in a controlled manner and allows teachers to run the same tests on a set of students' assignments. The component accepts any type of files as assignment and the teacher has a great flexibility in specifying how and what aspects of students' solutions have to be tested (like coding style, successful compiling, and so on). Furthermore, Testovid incorporates hints and advices into the testing reports, enabling teachers to give students comprehendible feedback about their programming solutions [Pribela et al. 2012]. This feature of the system inspired us to try to incorporate also software metrics to provide students with rich information about fallacies of their solutions and additionally evaluate the quality of student programs. It represents a good starting point

[Pribela et al. 2012] in the direction of development of appropriate pedagogical software metrics and to utilize software metrics in the assessment process of the student solutions to programming assignments.

5. CONCLUSIONS

Nevertheless the fact that software metrics are a well known way to measure the quality of software, existing automated assessment systems that have adopted them are still rare. It is also questionable if widely speeded software metrics like: Halstead metrics, McCabe cyclomatic complexity and some other NDepend metrics are common and useful static metrics for computer science education purposes. Our initial attempts to apply software metrics in programming courses and gained experiences indicate that it is useful and even necessary to develop and apply in everyday teaching specific pedagogical software metrics. Usage of such metrics could help students to develop systematic and higher-quality programs starting from introductory programming courses and further through other programming courses during their faculty education. Such approach could prepare them for future jobs and real-life software development where application of software metrics is getting unavoidable. So, in our future work we are going to put significant effort on developing specific pedagogical software metrics.

REFERENCES

Dennis M. Breuker, Jan Derriks, and Jacob Brunekreef. 2011. Measuring static quality of student code. In *Proc. of the 16th annual joint conference on Innovation and technology in computer science education* (ITiCSE '11). ACM, New York, NY, USA, 13-17.

Rick Giles and Stephen H. Edwards. 2011. Checkstyle. Available online [Last access on 24 July 2013]: http://bluejcheckstyle.sourceforge.net/#overview

Arnold L. Patton and Monica McGill. 2006. Student portfolios and software quality metrics in computer science education. *J. Comput. Sci. Coll.* 21, 4 (April 2006), 42-48.

Ivan Pribela, Mirjana Ivanović, and Zoran Budimac. 2009. Svetovid – interactive development and submission system with prevention of academic collusion in computer programming, *British Journal of Educational Technology* 40, 6, 1076-1093.

Ivan Pribela, Gordana Rakić, and Zoran Budimac. 2012. First Experiences in Using Software Metrics in Automated Assesssment. In *Proc. of the 15th International Multiconference on Information Society (IS)*, 250-253.

Tom Schorsch. 1995. CAP: an automated self-assessment tool to check Pascal programs for syntax, logic and style errors. ACM *SIGSCE Bulletin* 27, 1, 168-172.

Michael Steinhuber. 2008. Class Card – A Better UML Extension. Available online [Last access on 24 July 2013]: http://klassenkarte.steinhuber.de/index_en.html

Nghi Truong, Paul Roe, and Peter Bancroft. 2004. Static analysis of students' Java programs. In *Proc. of the Sixth Australasian Conference on Computing Education - Volume 30* (ACE '04), Raymond Lister and Alison Young (Eds.), Vol. 30. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 317-325.

Using Object Oriented Software Metrics for Mobile Application Development

GREGOR JOŠT, JERNEJ HUBER AND MARJAN HERIČKO, University of Maribor

Developing and maintaining software for multiple platforms can be challenging. So, scheduling and budget planning, cost estimation, software debugging, software performance optimization etc. is required. In traditional software, this can be achieved using software metrics. The objective of our article was to examine whether the traditional software metrics are appropriate for measuring the mobile applications' source code. To achieve this goal, a small-scale application was developed across three different platforms (Android, iOS and Windows Phone). The code was then evaluated, using the traditional software metrics. After applying the metrics and analysing the code, we obtained comparable results, regardless of the platform. If we aggregate the results, we can argue that traditional software metrics can be used for mobile applications' source code as well. However, further analysis is required, in light of a more complex mobile application.

Key Words and Phrases: Software product metrics, object-oriented metrics, mobile development, multi-platform mobile development

1. INTRODUCTION

The future of computing is moving towards mobile devices. It is suggested that by the end of 2013, people will purchase 1.2 billion mobile devices. As such, they will replace PC's as the most common method for accessing the Internet [Gordon 2013]. Currently, the two most popular platforms are Google's Android and Apple's iOS. There are, however, several other platforms, which add to the market fragmentation. Among those, Windows Phone is considered to be the most promising. Besides, mobile application development has also become an important area of scientific studies.

However, developers face several challenges when developing mobile applications. First, the range of mobile platforms requires the knowledge and experience with the different programming languages as well as software development kits (hereinafter referred to as SDK). This consequentially results in a duplication of development effort and maintenance costs [Nebeling et al. 2013]. Second, developers need to consider the physical limitations of mobile devices. Third, due to the low selling price of the mobile applications, developers need to reduce the development costs.

On the other hand, if we want to increase reusability, decompose problem into several easily understood segments or enable the future modifications, object-oriented programming metrics (hereinafter referred to as OO metrics) need to be considered. They represent a dependable guidelines that may help ensure good programming practices and write a reliable code. Such metrics have been applied and validated numerous times for the development of traditional software. In regard to the development of mobile applications, there is a lack of studies that would investigate the measurement of mobile application's software code. However, several attempts were already made in applying existing OO metrics to mobile applications, but were not empirically validated. Thus, based on the previous research we conducted the following hypothesis:

H_{01}: *Using the object-oriented metrics for mobile application does not differ from using object-oriented metrics for desktop or web applications.*

H_{A01}: *Using the object-oriented metrics for mobile application differs from using object-oriented metrics for desktop or web applications.*

Author's address: G. Jošt, UM-FERI, Smetanova ulica 17, 2000 Maribor; email: gregor.jost@uni-mb.si; J. Huber, UM-FERI, Smetanova ulica 17, 2000 Maribor; email: jernej.huber@uni-mb.si; M. Heričko, UM-FERI, Smetanova ulica 17, 2000 Maribor; email: marjan.hericko@uni-mb.si

In: Z. Budimac (ed.): Proceedings of the 2nd Workshop of Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA), Novi Sad, Serbia, 15.-17.9.2013, published at http://ceur-ws.org

However, there are also other mobile specific aspects that need to be considered. The number of users of mobile applications is usually dependent on both the quality of the application itself and the targeted platforms, for which the application was developed. Ideally, it would be best to develop each application for all platforms simultaneously without any adjustment of the source code. To this end, many tools for multi-platform development have been introduced, which support the "build once, deploy everywhere" ideology, but are not yet perfect. Mobile applications, developed in native languages are still preferred among the end users. But, as already stated, developing a mobile application for several platforms is a time and cost consuming process. The development of mobile applications for multiple platforms requires different environments and knowledge of specific programming languages, depending on the platform. Each platform has its own set of functionalities, rules and requirements, thereby limiting the interoperability of applications and their simultaneous development. Thus, we investigated if and how the results of metric analysis across different platforms differ in the scope of functionally equivalent application. In this light, the following hypothesis was formed:

H_{02}: *The results of metric analysis of functionally equivalent applications across different platforms do not differ.*

H_{A02}: *The results of metric analysis of functionally equivalent applications across different platforms differ.*

In order to reject or fail to reject both null hypothesis, with the help of colleagues, we implemented an equivalent applications for each of the chosen mobile platforms. The research included: 1) the development of said applications, 2) a comparison of the development between the mobile platforms and 3) metric analysis of the mobile application. We organized our research, and consequentially this article, as follows.

The second section introduces the research foundation. The third section reviews previous work regarding the challenges of developing applications for multiple platforms as well as metrics for measuring the mobile applications' source code, with the focus on the latter. The forth section presents the details of the application development on different platforms. We defined the basic functionalities, abstract design of the user interface and summarized the business logic. The fifth section presents the results of the metric analysis of the applications, whereas the last section interprets the results, reviews the limitations and implications, as well as defines future planned activities within the article's topics.

2. RESEARCH FOUNDATION

In general, software product metrics can be divided into two categories: 1) internal, which are seen usually only to development team (e.g., size, complexity and style metrics) and 2) external metrics (product non-reliability, functionality, performance and usability metrics), which assess properties, visible to the users of a product [North Carolina State University 2005]. In this light, our research is focused on OO metrics, which are one of the internal product metrics that can be used to evaluate and predict the quality of the software.

There is a vast body of empirical results, which support the theoretical validity of such metrics. The validation of these metrics requires a demonstration that the metrics measure the right concept (e.g. a coupling metric measures actual coupling). It is also important that the metrics are associated with an important external metrics, such as reliability or fault-proneness.

OO metrics are generally accepted as software quality indicator for desktop software. They provide automated and objective way to get concrete information regarding the source code [Franke and Weise 2011]. Such analysis with the purpose of quality evaluation continues to increase in popularity in wide variety of application domains.

Recently, mobile devices experienced a significant gain in performance and the development of mobile applications is getting more similar to desktop ones in light of the programming concepts and techniques.

Regarding the metrics for mobile applications, several suggestions were already made. OO metrics like McCabe Cyclomatic Complexity, Weighted Methods per Class or Lack of Cohesion in Methods have already been used to analyze the source code of software for mobile devices [Franke and Weise 2011].

On the other hand, external metrics, such as flexibility, which is a quality requirement for software, are also very important in the mobile domain. Environment of mobile devices can change often and suddenly (loss of mobile signal, different light conditions, etc.). Another important quality attribute is adaptability, which represents the ability of software to adapt itself to new underlying systems (e.g. usage of Galileo instead of GPS). Data persistence is also a quality requirement factor, since mobile devices can run out of memory, in which case the applications should be able to properly store their states. Because quality requirements for mobile software differ from their counterparts in desktop domain, the derivation of these qualities to source code metrics must also be done in a different way. So, metrics have to be evaluated in the mobile world and weighted correspondingly before their usage.

3. RELATED WORK

3.1 Challenges and comparison of the development of mobile applications for multiple platforms

When developing mobile applications in general, we are facing distinctive challenges. [Dehlinger and Dixon 2011] have identified the following:

1. Different sizes and resolutions of displays should be considered
2. Before the development of applications it is necessary to consider a variety of platforms, different hardware manufacturers, different development approaches (native, web or hybrid) and devices (phones, tablets).
3. Mobile applications are context-dependent, and they do not present a static context (in contrast to traditional platforms and applications).
4. Because of the dynamic environment and context it may not be possible to fully satisfy functional requirements. In this case, it is desirable that application is self-adaptive and thus can provide at least incomplete functionality rather than none.

Similar conclusions were also made during the development of the official IEEE application for iOS and Android platforms. Developers have identified the following specific challenges which are independent of the chosen platform [Tracy 2012]:

1. Because of the mobile device, the speed of the network can vary, since it depends on device's position.
2. Unavailability of the network; mobile applications need to be custom made for execution in offline mode or be able to save the state before exiting.
3. Operating system, dependent on the hardware. For example, Android can behave differently on a variety of different hardware specifications. Developers need to take this into the account.
4. Different screen sizes and different sensor support.
5. Due to the large number of different mobile devices, it is difficult to comprehensively test the application.

One of the major challenges of mobile application development is the development of user interface, as was address by [Larysz et al. 2011], where a comparison and testing of user interfaces has been done for iOS, Android and Windows Phone. Another issue is also choosing a target platform for application development [Ohrt and Turau 2012]. Unlike desktop applications, where 90% of users use Windows OS, mobile application developers must consider a number of current mobile platforms to reach the same percentage of users.

3.2 Software metrics

In this section, the related work of mobile metrics is presented. We have analyzed the current state of measuring the mobile applications' source code. Based on the related work we can conclude, that metrics for the analysis of mobile applications can be divided into two groups, namely, metrics that address the popularity of applications (number of downloads, installations) and software product metrics that measure both internal and external aspects of application development. As part of the related work, we focused on the internal software product metrics.

In the article [Kumar Maji et al. 2010], authors analyzed the number of bugs and errors on platforms such as Android and Symbian. Besides analyzing the bugs in aforementioned platforms, authors also measured their complexity, using the metrics of McCabe's cyclomatic complexity and the lines of code.

As part of the development of the middleware software, which is located above the Android platform and is responsible for implementation and maintenance of context-aware applications, the authors applied metrics to compare the conventional and their own approach for implementing mobile applications [David et al. 2011]. In addition to the software abstraction their proposed solution also offers composition, reuse and dynamic installation. Their approach was also qualitatively analyzed in the scope of a prototype implementation of the context-aware instant messaging application. The application checks user's movement and if it detects increase in speed, it warns the user. The said application was developed in two ways: simple native Android implementation and implementation using authors' proposed approach. The most important components of the latter are Context Manager Service (CMS), which collects and processes context data and Situation Evaluation Engine (SEE), which is basically a Java library that enables the developer to define their own context situation. The analysis of both applications was conducted with the use of the following metrics:

- *Number of external classes*, which represents the number of system classes that had to be included in the implementation. Due to the architecture of CMS and SEE approach, this number was always 2, as opposed to native Android development, where this value was 8.
- *The number of new methods* represents all the methods that must be implemented in the context of API interfaces. In case of CMS and SEE approach, this number was always 1, whereas for native Android development, this number was 8.
- *External constants* represents the number of constants for error code. Since SEE only returns error when information is not available, this number was always 1. Android had 5 external constants.
- *The number of concepts to learn* is limited to the platform-specific concepts such as Looper, Listener, Android Services, GPS activation etc. In the case of SEE, only Situation and SituationRule are needed, so this number is always 2. Again, Android had a higher number of concepts to learn (7).
- *The number of new lines of program code* was significantly higher for Android (167 vs. 59).
- *The time required for the development of application* (48 hours for Android and 18 hours for CMS and SEE approach).
- *The time required to test the application* (12 hours for Android and 2 hours for CMS and SEE approach).

The review of the related work shows that there are only a few articles that analyze the use of metrics that measure the programming code of mobile applications. The reason for this may be due to the fact that we can use conventional software metrics for the development of mobile applications, which we will examine in details in the following chapters.

4. IMPLEMENTATION OF PROTOTYPE MOBILE APPLICATIONS FOR MULTIPLE PLATFORMS

In this chapter we will introduce the 1) prototype mobile application, 2) supported platforms and 3) process of multiple platform development.

4.1 Prototype mobile application

Our prototype mobile application represents a mobile version of a video store. It enables users to rent and search for movies. The administrators can read the barcodes of the movies for the sake of renting and returning them. Application has been designed in a way that includes the advanced and most commonly used concepts of mobile platforms, among which are location services, usage of maps, web services and sensors. The goal of the prototype mobile application was to develop an application on different platforms, which will ensure the native look and feel of each platform while providing the same functionalities and similar appearance.

4.2 Selected platforms

The research was limited to the following platforms: iOS, Android and Windows Phone. The reasons for choosing these platforms are as follows: Android and iOS had a more than 50% combined market share in 2011 [Gartner 2011], which is also reflected in the number of applications that are available for said mobile platforms. Half a million applications for iOS [Apple Inc. 2013] and close to 400 thousand for Android [AppTornado GmbH 2012] are a clear indication that these two platforms are popular among the users and developers. Windows Phone is the youngest of the modern mobile platforms, which is speculated to have at least 20% market share in 2015 [Gartner 2011]. The market for Windows Phone applications is also rapidly growing and has reached 50 thousand applications in a little more than a year [Blandford 2011].

Furthermore the following programming languages were used. The Windows Phone application was developed using C# programming language combined with XAML markup language. Visual Studio 2010 IDE was used. The Android application was built in Eclipse IDE, using Java programming language, combined with XML. Finally, iOS application was developed using Objective-C programming language in Xcode IDE.

4.3 Development of the prototype application

When developing applications for multiple platforms, a goal is not only to achieve quick and easy to use solutions, but also to preserve the unique user experience (look and feel), depending on the platform [Hartmann et al. 2011]. In the following subchapters we defined two aspects, which have to be unified when developing an application for different platforms, namely building a common user interface and business logic.

4.3.1 Building a common user interface

When building a user interface for cross-platform application, we have to take into the consideration advantages and disadvantages of specific platforms. When trying to achieve the same user experience (regardless of the mobile platform) we have to be aware of these restrictions or the application can result in an unnatural user experience. The visualization of the graphical elements can also impact the user experience. Although Android and iOS are somewhat visually similar, this does not apply to Windows Phone platform. The latter includes the "Windows 8 style UI", which significantly differs from others.

After we analyzed graphical elements of the aforementioned platforms, we found out that out of 26 identified, there are 19 elements on all three platforms, which have the same functionality. By using these elements we are guaranteed the same functionality of the user interface, as well as visual consistency with design guidelines of platforms. Visual coherence defines not only the correct visual form of graphical elements, but also compliance with the color schemes that are imposed by the platform. While iOS has no specifications regarding the color schemes, with the arrival of Android 4.0, Google indicated the visual guidelines for developers. Windows Phone on the other hand promotes a dynamic color changing in the applications, which depends on the color scheme the user has set for the entire operating system [Microsoft 2010].

Figure I represents one of the final screens of our applications on Windows Phone, Android and iOS platform, respectively.

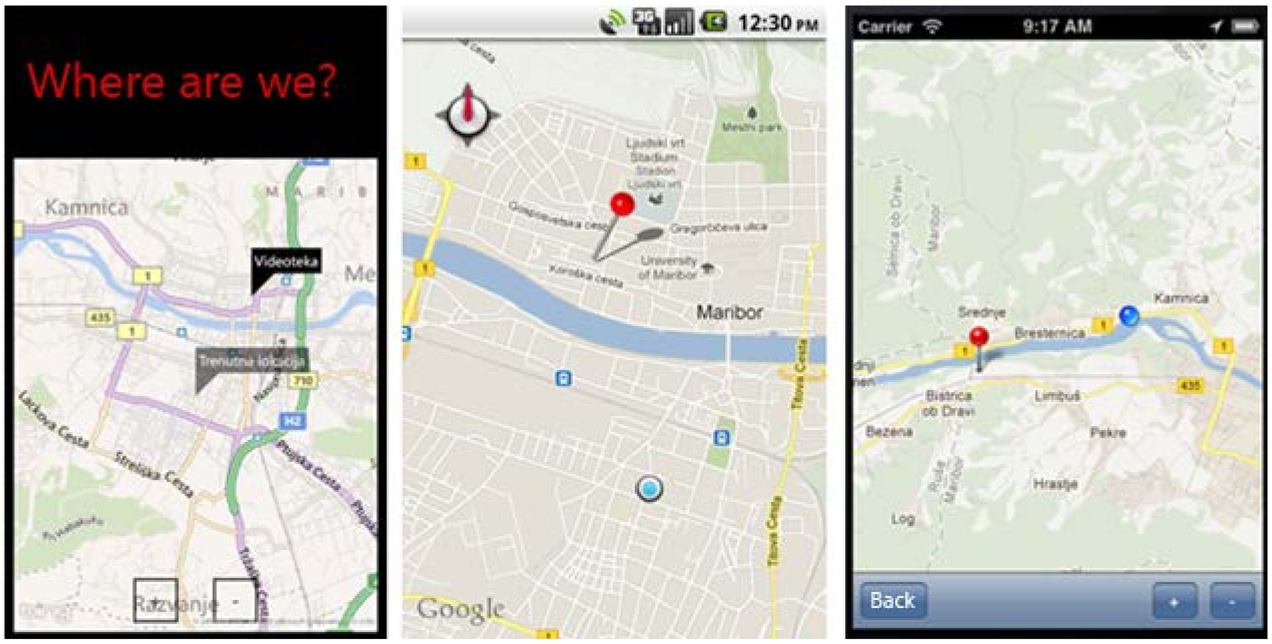


Figure I: Sample screen of the mobile application across different platforms (Windows Phone, Android and iOS, respectively)

4.3.2 Business logic

Developing applications for multiple platforms can be the cause of code duplication. If we want to maximize the reuse of code across platforms, we have to move it to the server side. In determining which part we want to move to the server, it is important to consider several factors. So, it is advisable to move the business logic to the server in the following cases:

- A particular operation consumes a lot of computing resources.
- The consistency of data is required.
- Business logic changes often, in which case we do not need to change client applications and maintain multiple versions of the business logic for older clients.
- We operate with a larger amount of data. In this case, it might not be practical to keep it entirely on the client.

However, there are situations where business logic should be on the client side. Such scenarios are as follows:

- When client has to transfer a large amount of data (for example, resizing the images).
- There is a need for off-line mode.
- There is a need for faster computing.

Business logic on the server side is usually exposed in the form of web services. This mode thus brings a new architecture: a web service-client. There are currently two popular methods of exposing web services: REST and SOAP. If we are developing a mobile applications for all three platforms, we have to use REST web services, since SOAP services are not (officially) supported on iOS and Android. REST is not a protocol but rather an architecture, which uses HTTP protocol to transfer data. So, only four methods (GET, POST, PUT, and DELETE) out of the nine are available. Regarding the format of the data, we have two options: XML or JSON [Rodriguez 2008].

Table I represents the distribution of files across the selected platforms, based on their purpose in the application structure.

Table I: Distribution of files by purpose across different platforms

Type of the files	Android	iOS	Windows Phone
Business logic	10	8	8
User interface	29	23	12
Utility files	21	5	3
Other files	3	6	5
Together	63	42	28

As is evident from the table above, Windows Phone platform uses the most files for graphical user interface (12 file out of 28 files). Second most of other files are used for business logic (8 files). In case of iOS, the majority of files is again used for graphical user interface (23 files out of 42 files). They are followed by files for business logic (8 files). Similarly, in case of Android platform, majority of files are used for the GUI purposes (29 files out of 63 files), followed by utility files (21) and business logic (10).

However, it should be noted that the server side logic was not part of the analysis.

5. METRIC ANALYSIS OF PROTOTYPE MOBILE APPLICATIONS

In this chapter, selected metrics are introduced. Furthermore, the selected plugins and tools for static code analysis are defined and the results of the metric analysis are presented.

5.1 List of selected metrics

For the sake of evaluating the code quality, besides other traditional software metrics, we primarily used the Chidamber and Kemerer (CK) metrics. The selected metrics along with the brief descriptions are provided below [North Carolina State University 2005], [Rosenberg 1998], [Chidamber and Kemerer 1994].

The first metric, depth of inheritance tree (hereinafter referred to as DIT), is defined as the maximum length from the node to the root of the tree. DIT is predicted on the following assumptions:

- The deeper a class in the hierarchy, the greater the number of methods it will most likely inherit.
- Deeper trees involve greater design complexity (due to the fact that more classes and methods are present).
- Deeper classes in the tree have a greater potential for reusing inherited methods, which is good for potential code reuse.

Number of children (NOC1) is the number of direct subclasses for each class. Classes that have a large number of children are considered difficult to modify and are also considered more complex and fault-prone.

Lack of Cohesion in methods (LCOM) represents the number of disjoint or non-intersection sets of local methods. A higher LCOM value represents a good class subdivision, whilst a low cohesion increase complexity and subsequently increasing the possibility of errors during the development of software.

Weighted Methods per Class (WMC) represents the McCabe's cyclomatic complexity. WMC is dependent on number of Methods (NOM), which indicates complexity of an individual class. Both higher WMC and NOM of a class suggest, that such a class is more complex than its peers and more application specific, limiting the potential reuse of code.

Number of classes (NOC2) is significant when comparing projects with identical functionality. Projects with more classes are perceived as being better abstracted.

Similarly to NOC2, lines of code (LOC) metric is also significant when comparing similar projects. LOC indicates the approximate number of lines in the software code. A higher LOC value might indicate that a type or method is doing too much work. Subsequently, it might also indicate that the type or method could be hard to maintain.

Cyclomatic complexity (CC) indicates how hard software code may be to test or maintain and how likely the code will produce errors. Generally, we determine the value of cyclomatic complexity by counting the number of decisions in the source code.

The described metrics are summarized in Table II in light of preferred values (either high or low) for better code quality.

Table II: Summary of selected metrics

Metric	**Desired value**
Depth of the inheritance tree	Low
Number of Children	Low
Lack of Cohesion in methods	Lower
Weighted methods per class	Low
Number of methods	Low
Number of Classes	Higher
Lines of Code	Lower
Cyclomatic complexity	Low

5.2 Results of the metric analysis

The metrics for analyzing prototype mobile applications were gathered by using the following software or plugins, depending on the platform:

- In case of Windows Phone, we used "NDepend", a tool for static code analysis for .NET applications [Smacchia 2013].
- For analyzing Android application source code, we used "Metrics2" plugin, which is available for Eclipse integrated development environment (IDE) [Sauer 2002].
- Xcode, which was used for developing the iOS application, does not enable a metric analysis of applications, it only notifies the developer in case of errors in the code. Thus, we used "Understand Metrics", a tool for static code analysis, which, among others, enables the analysis of Objective-C [Scientific Toolworks, Inc 2013].

Table III represents the results of the analysis of the source code. Cells with the gray background represent the least desirable values among platforms.

Table III: Results of metric analysis of prototype mobile applications

Metrics	**Android**	**Windows Phone**	**iOS**
DIT (average)	**2,237**	0,5	0,033
NOC1 (average)	**0,475**	0,055	0,033
LCOM (average)	0,186	0,452	**0,764**
WMC (average)	7,242	**15,174**	4,494
NOM	**249**	143	191
NOC2	**59**	18	49
LOC	2707	554	**3482**
CC (average)	1,72	**1,91**	1,15

As is evident from the Table III, Android has the highest DIT. However, this is mostly due to the fact that Android platform requires the inheritance of certain system classes, which already have a high DIT by default. The maximum value of DIT in case of Android was six. An example of such a class represents the screen for displaying the map. This particular class inherits the following chain of classes: *MapActivity* ← *Activity* ← *ContextThemeWrapper* ← *ContextWrapper* ← *Context* ← *Object*. In case of Windows Phone and iOS, we used an external tool for static code analysis. These tools do not take into the account the inheritance of system classes within platform libraries.

Differences between platforms can be once again seen in case of NOC1, which was the highest in the case of Android platform. However, "Metrics" plugin takes into the consideration not just classes, but also the interfaces. Fundamentally, the default Windows Phone and iOS structure does not include as many interfaces as Android.

LCOM is the lowest in the case of Android platform, mainly due to the GET, SET and toString methods. This is also reflected when viewing the metrics for packages and individual classes.

WMC was calculated using the following formula:

$$WMC = \left(\frac{NOM}{NOC2}\right) * CC$$

WMC has the highest value in case of Windows Phone platform, which means that there are more methods per class than in case of other platforms. It should be noted that the complementary metric NOM indicates that application, developed for Android platform has the most methods. However, in case of Android platform, these methods are divided into several classes. The increase in WMC in case of Windows Phone is mainly due to the frequency of asynchronous service calls and rewriting of the methods to navigate between pages of application.

When interpreting the LOC it is important to consider the programming language, SDK and related set of functionalities (e.g. libraries). Thus, in case of Windows Phone, the .NET framework includes advanced libraries which is also reflected in LOC metric. Parsing XML with LINQ and inclusion of attributes in entities significantly decreases the LOC metric value. Conversely, in case of Android platform, we have GET and SET methods, which additionally increase the LOC metric value. CC metric focuses on the number of branches and loops. CC metric had the lowest value in case of the iOS platform. Both Android and Windows Phone 7.5 had the higher CC metric value, mostly due to the usage of dynamic screens, whereas in case of iOS the navigation between screens is simplified using the iOS Storyboard. Since we were able to apply the metrics to mobile application's source code and gathered sound results, we failed to reject the null hypothesis H_{01}. However, since the results were not similar among the selected platforms for the functionally equivalent applications, the second null hypothesis H_{02} was rejected in favor of H_{A02} (see Table III).

6. CONCLUSION

The objective of our goal was to examine whether the traditional software metrics are appropriate for measuring the mobile applications' source code and to this end, a small-scale application was developed across different platforms. The code was then evaluated by using the traditional software metrics. After analyzing the results, we obtained sound results, regardless of the platform. In this section, we will highlight the limitations of our research, which will be followed by implications and future work. Finally, a summary of this article will be presented.

6.1 Validity evaluation

When interpreting our results, the following limitations should be taken into the account. Firstly, we have built a small-scale applications for different platforms. Larger-scale application might produce different results. Secondly, only three of the existing mobile platforms have been used. Object-oriented metrics might not be appropriate for other mobile platforms. Thirdly, out of the selected platforms, the following versions were used: iOS 5.0, Windows Phone 7.5 and Android 2.1. More recent versions of these platforms might produce different results when analyzing the source code. Fourthly, the scope of the selected metrics was limited by capabilities of existing plugins and other third-party tools for static code analysis. Other metrics might not be appropriate for mobile applications. Fifthly, we did not conduct any advanced statistical analysis, which would confirm or reject statistically significant differences of metrics' values between platforms. Sixthly, prototype mobile applications were developed by three different programmers, one for each of the selected platforms. Had the same programmer develop mobile applications for the said platforms, the results of the metric analysis might differ. And finally, the metrics were not validated within the scope of mobile environment, but were merely applied to our prototype mobile applications.

6.2 Implications and future work

Although the existing literature suggested the usage of OO metrics in the scope of mobile application development, none had any empirical data. Also, none of the existing literature compared results of applying metrics between functionally equivalent mobile applications on different platforms. In this light, our results present the starting point for detailed analysis of applying OO metrics to the domain of mobile application development. The most important finding therefore was the fact that we successfully applied OO metrics to mobile domain. Also, our results suggest that the difference between mobile platforms in the sense of applying the metrics to the source code of mobile application is present. Thus, an application with the similar functionalities, graphical user interface and used sensors might produce different metrics' values when considering the differences between the platforms. So, when applying OO metrics to mobile domain, those differences should be considered in the future research.

In the future we plan to conduct an empirically validated and statistically detailed analysis of OO metrics in the scope of a large-scaled mobile application development. We will include more existing mobile platforms (e.g. Blackberry) as well as multi-platform frameworks.

6.3 Summary

The objective of our article was to examine whether the traditional software metrics are appropriate for measuring the mobile applications' source code. A small-scale application was then developed across three different platforms (Android, iOS and Windows Phone). We evaluated the code by using the traditional software metrics. After applying these metrics and analyzing the code, we obtained sound results on all the platforms. If we aggregate the results, we can argue that traditional software metrics can be used for mobile applications' source code as well, which was the notion of the first null hypothesis. However, since the results of metrics' analysis of functionally equivalent application were slightly different between platforms, the second null hypothesis was rejected. Further detailed statistical analysis is required, in the scope of a more complex, large-scale mobile applications.

ACKNOWLEDGEMENTS

The authors wish to thank Aleš Černezel, Mitja Krajnc and Boris Ovčjak for their assistance in preparing the prototype mobile applications as well as for helping with writing of the article.

REFERENCES

Apple Inc., 2013. The AppStore. Apple.

AppTornado GmbH, 2012. Number of available Android applications. AppBrain.

Blandford, R., 2011. Windows Phone Marketplace passes 50,000 apps. All About Windows Phone.

Chidamber, S.R. and Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 20(6), pp.476–493.

David, L., Endler, M., Barbosa, S.D.J. and Filho, J.V., 2011. Middleware Support for Context-Aware Mobile Applications with Adaptive Multimodal User Interfaces. In 2011 4th International Conference on Ubi-Media Computing (U-Media). 2011 4th International Conference on Ubi-Media Computing (U-Media). pp. 106–111.

Dehlinger, J. and Dixon, J., 2011. Mobile Application Software Engineering: Challenges and Research Directions. In Santa Monica, CA, USA.

Franke, D. and Weise, C., 2011. Providing a Software Quality Framework for Testing of Mobile Applications. In 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST). 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST). pp. 431–434.

Gartner, 2011. Gartner Says Android to Command Nearly Half of Worldwide Smartphone Operating System Market by Year-End 2012. Gartner Newsroom.

Gordon, A.J., 2013. Concepts for mobile programming. In Proceedings of the 18th ACM conference on Innovation and technology in computer science education. ITiCSE '13. New York, NY, USA: ACM, pp. 58–63.

Hartmann, G., Stead, G. and DeGani, A., 2011. Cross-platform mobile development.

Kumar Maji, A., Hao, K., Sultana, S. and Bagchi, S., 2010. Characterizing Failures in Mobile OSes: A Case Study with Android and

Symbian. In 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE). 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE). pp. 249–258.

Larysz, J., Němec, M. and Fasuga, R., 2011. User Interfaces and Usability Issues Form Mobile Applications. In V. Snasel, J. Platos, & E. El-Qawasmeh, eds. Digital Information Processing and Communications. Communications in Computer and Information Science. Springer Berlin Heidelberg, pp. 29–43.

Microsoft, 2010. UI Design and Interaction Guide for Windows Phone 7, version 2.0.

Nebeling, M., Zimmerli, C. and Norrie, M., 2013. Informing the design of new mobile development methods and tools. In CHI '13 Extended Abstracts on Human Factors in Computing Systems. CHI EA '13. New York, NY, USA: ACM, pp. 283–288.

North Carolina State University, 2005. An Introduction to Object-Oriented Metrics.

Ohrt, J. and Turau, V., 2012. Cross-Platform Development Tools for Smartphone Applications. Computer, 45(9), pp.72–79.

Rodriguez, A., 2008. RESTful Web services: The basics. IBM.

Rosenberg, L., 1998. Applying and Interpreting Object Oriented Metrics. In Software Technology Conference. NASA Software Assurance Technology Center (SACT).

Sauer, F., 2002. Eclipse Metrics plugin.

Scientific Toolworks, Inc, 2013. Understand Metrics.

Smacchia, P., 2013. NDepend.

Tracy, K.W., 2012. Mobile Application Development Experiences on Apple #x2019;s iOS and Android OS. IEEE Potentials, 31(4), pp.30–34.

Beta Testing of a Mobile Application: A Case Study

MATEJA KOCBEK AND MARJAN HERIČKO, University of Maribor

Beta testing is the last stage of testing, and normally involves sending the product for beta testing and real-world exposure outside the company. Beta testing is often preceded by a round of testing called alpha testing. Beta testing can be considered a form of external user acceptance testing. Software in the beta phase will generally have many more bugs in it than completed applications, as well as speed and performance issues that may still cause crashes or data loss. Beta testing is the first opportunity to get real feedback from target customers. The launch of a mobile application is especially crucial because it is the single biggest opportunity to get an application discovered in the mobile markets. In this article, the beta testing of mobile applications is presented. Our aim is to identify the optimal number of testers, who can reveal the majority of errors and mistakes during beta testing. The findings were obtained through the case study research method.

Key Words and Phrases: alpha testing, beta testing, software testing, acceptance testing

1. INTRODUCTION

The software development life-cycle has several phases, which should be conducted in the following order: analyse requirements, design, development, integration and testing, deployment and maintenance. We will focus on the testing phase.

However, the successful introduction of integrated systems into businesses greatly depends on whether we trust the system or not. Trust in the quality of the software-based system is determined by many factors, such as: completeness, consistency, maintainability, security, safety, reliability, and usability. However, during the development of a software-based system, there are many opportunities to find errors in the different phases of the software development lifecycle. Testing is commonly applied as the predominant activity to ensure high software quality, providing a wide variety of methods and techniques to detect different types of errors in software systems [Budnik 2012].

Software bugs will almost always exist in any software module: the complexity of software is generally intractable and humans only have a limited ability to manage complexity. Thus, design defects can never be completely ruled out, especially in complex systems [Pan 1999].

Testing is usually performed for the following purposes: (1) to improve quality, (2) for verification and validation, and (3) for reliability estimation. *To improve quality* refers to the conformance of the specified design requirements. Being correct, the minimum requirement of quality, means performing as required under specified circumstances. Debugging, a narrow view of software testing, is intensively performed to discover design defects by the programmer. The imperfection of human nature makes it almost impossible to make a moderately complex program correct the first time around. Another important purpose of testing is *verification and validation*. Within the process of verification and validation, testing can serve as a metric. Software reliability has an important connection with many aspects of the software, including structure and the amount of testing it has been subjected to. Testing can also serve as a statistical sampling method to gain failure data for *reliability estimation* [Pan 1999].

As previously mentioned, the process of testing software is very complex. For this reason there are many types of software testing. One of the formal types of software testing is alpha testing. Usually it is performed by potential users/customers or an independent test team at the development site. Alpha testing is conducted before taking the software to beta testing [Pradhan 2012].

In our paper, the actual beta testing of mobile applications will be presented. Due to the reliance on alpha and beta testing, these two types of testing will be presented as well. The main aim of the article is

Author's address: M. Kocbek, Institute of Informatics, Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova ulica 17, 2000 Maribor, Slovenia; email: mateja.kocbek@um.si; M. Heričko, Institute of Informatics, Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova ulica 17, 2000 Maribor, Slovenia; email: marjan.hericko@uni-mb.si.

In: Z. Budimac (ed.): Proceedings of the 2nd Workshop of Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA), Novi Sad, Serbia, 15.-17.9.2013, published at http://ceur-ws.org

to identify what the optimal number of testers is to reveal the majority of defects in a mobile application. In the second chapter, the background of the case will be briefly described. Some facts about mobile applications, alpha and beta testing, and the case study will be given. The third chapter is about the case: all the details of the beta testing of the actual project are given and research question is discussed. In the end, we present our conclusions.

2. BACKGROUND

In this chapter, the background of all content areas is given.

2.1. Mobile application

The era of mobility and mobile applications has arrived, and it demands the rapid development and delivery of high-quality solutions. Customers expect an exceptional mobile experience and competition in the mobile market gives them a wide range of choices. The consumer world has driven unprecedented changes in IT and now IT needs the practices, delivery models, and management solutions to make it work. The unprecedented increase of mobile app development has brought a new set of challenges that engineering and quality assurance (QA) teams need to overcome in order to keep up with this rapid rate of growth [HP 2012].

Mobile application is software designed to use it on smart devices. The mobile applications are available to the end users through mobile markets. Mobile applications are actual applications that are downloaded and installed on mobile devices, rather than being rendered within a browser. The application may pull content and data from the internet [Summerfield 2013]. A mobile application is software that runs on a device that can connect to a wireless carrier network and has an operating system that supports standalone software.

There are two types of mobile applications. The first are native applications. This is an application that runs on a handheld device (phone, tablet, e-reader, iPod Touch) which has a “smart” operating system which supports standalone software and can connect to the internet. Usually native applications are downloaded from app stores such as the Apple Store, Google Play, etc. A native application can only be “native” on one type of mobile operating system. This is why, for example, an iPhone application only works on iOS devices. Native applications are designed to run on a device’s operating system and machine firmware, and typically need to be adapted for different devices [MobiThinking.com 2013]. By contrast, a mobile web application is software that uses technologies such as JavaScript or HTML5 to provide interaction, navigation, or customization capabilities. These programs run within a mobile device’s web browser. This means that they are delivered wholly on the fly, as needed, via the internet; they are not separate programs that get stored on the user’s mobile device [Gahran 2011]. In a web application, or also a browser application, all or some parts of the software can be downloaded from the Web each time it is run. It can usually be accessed from any web-capable mobile device [MobiThinking.com 2013].

Mobile application development is also a special challenge. The nature of the mobile platform for which an application is developed requires a great deal of planning and consideration from the developer. Mobile devices have unique characteristics, such as: bandwidth, processor speed, screen size and resolution, memory consumption, battery life, and user input tools. These are issues that carry a certain level of importance in a desktop application as opposed to mobile applications [Mahmoud and Popowicz 2010].

Given the extreme pace and unpredictability of today’s mobile market, maintaining application quality has become a daunting task. The complexities of multi-platform development coupled with the ever-growing number of models, operating systems, screen sizes, and network technologies make it practically impossible to keep your mobile apps and services in sync with the ever-changing landscape of technology. To address these business and technological challenges, organizations need to implement a mobile testing strategy for the testing team [HP 2012].

2.2. Alpha and beta testing

Every company or organization follows its own software testing life-cycle. We focus on a specific stage of software testing called beta testing.

While the alpha testing is done on the side of the developer, beta testing is the first actual user test of software that provides enterprise software manufacturers with end-user usability and software functionality feedback. Beta testing begins in the last phase of the software development cycle and lasts until the final release of the software. Software developers select special users or user groups to validate software and provide tools to collect feedback. The purpose of beta testing is to enhance the product prior to general availability. Major software manufactures are focusing on improving the quality, acceptance and experience of enterprise software by promoting beta testing programs. It should be noted that there are no existing standards or models for beta software testing [Buskey 2005].

Beta testing is the most effective process to generate information about invalid or empty inserts and other similar scenarios. Beta testing is executed using a black box technique, with no direction or instructions from software development. Beta testing is also a process that extends the validation and testing phase during the software development life cycle. Beta testing strengthens the validation and testing phase and fosters the deployment phase by assuring the product is fully tested [Buskey 2005].

Beta testing is an essential component of the software validation and testing phase because of its immediate impact on the product being tested. It provides a platform to integrate real-world feedback into a product, prior to availability. This collectively improves the software usability and functionality, which lowers the potential cost and improves quality [Fine 2002]. The impact of beta testing on the software industry is important and global standards are required to manage this process [Buskey 2005].

2.3. Case study

Case study research is conducted in order to investigate contemporary phenomena in their natural context. That is, no laboratory environment is set up by the researcher, where factors can be controlled. Instead, the phenomena are studied in their normal context, allowing the researcher to understand how phenomena interact with the context. The selection of subjects and objects is not based on statistically representative samples. Instead, the research findings are obtained through the in-depth analysis of typical or special cases [Runeson and Höst 2008].

Case study research is conducted by iteration over a set of phases. In the design phase, the objectives are decided on and the case is defined. Data collection is first planned with respect to data collection technique and data sources. Methods for data collection include interviews, observation and the use of archival data. During the analysis phase, insights are both generated and analysed, e.g. through the coding of evidence from the findings to the original data. The report should include sufficient data and examples to allow the reader to understand the chain of evidence [Runeson and Höst 2008].

3. ABOUT THE PROJECT

3.1. Case study design

The focus of this study is the beta testing of mobile applications. Beta testing can be considered to be a form of external user acceptance testing. Software in the beta phase will generally have many more bugs in it than completed applications, as well as speed and performance issues that may cause crashes or data loss. Beta testing is the first opportunity to get real feedback from target customers, because the beta testing team is independent from the development team. Pre-beta testing should be noted too. Pre-beta testing is the phase between the alpha and beta testing phase, although there is no exact definition of this term.

The authors of this article worked on a project that is described through this case study. That is why the method for collecting data is descriptive. This was the only used method. In the study, only qualitative data is used and analysed.

Our research question is: How many testers are needed to reveal the majority of defects of a mobile application during beta testing?

There is a connection to existing literature, which uses basic mathematical principles to describe and summarize given facts. In study [Nielsen 2000], facts are given about how many users are needed and how many errors can be found during testing. The author's intention was to find the optimum number of users/testers to publish a stable mobile application, where 85% of errors are detected.

3.2. About the project

As previously mentioned, mobile applications can be developed during a relatively long process, called the software development process. From this perspective, they are just the same as regular desktop or web applications. The project from our case is a project developed at the Institute of Informatics, at the Faculty of Electrical Engineering and Computer Science, at the University of Maribor. The development process was nearly one year long. All phases of the software development process were carried out. There was a group for Android development, which was separate from the group for iOS development. Every group had three members – software developers. A very important part of the project was also the group for design, comprised of three designers. The internal group for testing had about five members, but there was also an external group of testers.

3.3. Performing beta testing

The project roughly described before had three versions. The first version was published in November 2012, the second in March 2013 and the third in July 2013. For every version of the application, comprehensive testing was necessary. Once the application was published, there was no way back. The process of internal testing was essential. The internal group for beta testing was receiving daily builds. The daily builds were reviewed in detail. All detected errors, deficiencies, inconsistencies and feature requests were logged with the system Flyspray [Flyspray 2012]. System enables to log several characteristics, like: reported version, place, severity, priority, name of the tester, etc. The system was used by the internal group for beta testing and the developers. This was the process on a daily basis. The external group for beta testing obtained the build every week. They had to report about their tests to the internal group for beta testing. They checked all existing defects and noted ones that did not arise before. At each stage before every release, the internal and external group had to report issues.

As mentioned before, the group for beta testing was essential for our article. Testers were performing automated and also manual testing on real devices and in different environments. The main goal of every group of testers was to identify as many defects as possible to ensure a standalone application. To ensure this condition, some essential resources were needed, the most important being human resources. How many people were actually needed? The source [Nielsen 2000] reveals that the best results come from testing with only 5 users/testers. The number of defects (X) found at testing with n users is:

$$X = N\,(1\text{-}(1\text{-}L)\ n), \qquad (1),$$

where N is the total number of detected problems in the design and L is the proportion of defects discovered while a single user is testing. The typical value of L is 31%, averaged across a large number of projects. Plotting the curve for L = 31% gives the following results: The most striking truth is that zero users give zero insights (Equation 1). As soon as data from a single test user is collected, the insights dramatically rise and almost a third of all defects are found. The difference between zero and even a little bit of data is astounding. The second user discovers some of the same things as the first user, so there is some overlap. People are definitely different, so there will also be something new that the second user does that was not observed by the first user. Thus, the second user adds some amount of new insight, but not nearly as much as the first user does. The third user will do many of the things already observed by the first or second user and even some things that have already been caught twice. Of course, the third user will generate a small amount of new data, even if it is not as much as the first and second users have. By adding more and more users, less and less defects are discovered because the same objects are repeated. After the fifth user, basically the same findings are discovered [Nielsen and Landauer 1993; Nielsen 2000].

The project was performed on the basis of described theory. The whole group for beta testing (internal and external together) consisted of 10 testers. Before testing, they received instructions regarding which functionality needed to be tested. With every functionality, black box testing was performed. Every tester performed a number of tests every day, wherein the Flyspray system was the main logging tool. Every defect detected by the tester was noted there. A tester had to be up-to-date with the situation on the Flyspray, because there was no need to duplicate defects.

In order to facilitate data analysis, we will concentrate on the last week before the last existing version of our mobile application was published. The discovered defects were classified according to whether they were a contextual or technical problem. All testers together found 39 defects (29 errors, 10 deficiencies and inconsistencies). According to the classification, 62% of all faults were technical problems while the rest were contextual. Duplicates were not included. To validate our theory, we first focused on the first 5 testers. The results are given in *Table 1*. Tester A found 17 errors. Tester B found 12 errors, in which 5 of them were new. Tester C found 3 new errors, tester D found 2 and tester E found 1 new defect. The number of new defects is declining fast. The biggest difference is between the first two testers. Tester A found 43 % of all defects found in this particular week.

Table 1: Number of defects of first five testers

	Number of defects	Number of new defects
Tester A	17	17
Tester B	12	5
Tester C	10	3
Tester D	7	2
Tester E	6	1

The rest of the testers from F to J found a minor number of defects (*Table 2*). But more than the number of defects are significant new defects. *Table 2* shows that the rest of the testers from F to J found only one new defect.

Table 2: Number of defects of second five testers

	Number of defects	Number of new defects
Tester F	5	1
Tester G	4	0
Tester H	3	0
Tester I	1	0
Tester J	1	0

Testers from A to E found altogether 97 % of all the defects found. This number represents the great majority of all defects found. In the report of the beta testing group, there were also a lot of defects, which were duplicates or not real defects. Beta users did not know all the requirements for software requested by the customer. This is the reason why some defects were not included in the analysis.

Our research confirmed numbers and facts from [Nielsen 2000] which we can easily answer the research question stated above. Five testers can reveal the majority of defects, while adding an additional tester does not add a crucial performance boost. Although a sixth tester can enhance the ratio of detected defects, this difference is not crucial. With the current project we revealed that the optimal number of testers when performing beta testing is five. With such a number of testers, work efficiency and the optimal consumption of resources can be ensured.

3.4. Related Work

Beta testing in the mobile domain is a relatively undiscovered area. Since mobile applications are different from traditional ones, they require different and specialized new techniques for testing [Kirubakaran and Karthikeyani 2013]. Beta testing is one of the phases where mobile applications are verified. Some basic facts about beta testing in general are given in [Buskey 2005]. Another type of testing of mobile applications is described in [Long 2010]. In [Muccini et al. 2012] directions for mobile testing automation are given. Similar content to that found in our article was not found.

CONCLUSION

We can conclude that a group for beta testing is essential in a stand-alone project that is waiting to be published. Another important perspective is also the need for iterations. The described process needs to be performed all over again, in order to gain effectiveness. This is especially important if the functionalities change or if the customer does not know exactly what to offer to the user. That is why the specifications for software are very important document for customers, developers and especially for testers, who ensure quality. Our case study can confirm that the optimal number of testers is roughly five. Adding more testers to a group does not ensure additional quality.

REFERENCES

BUDNIK, C., 2012. Software Testing, Software Quality and Trust in Software-Based Systems. *2012 IEEE 36th Annual Computer Software and Applications Conference*, pp.253–253.

BUSKEY, C.D., 2005. A Software Metrics Based Approach to Enterprise Software Beta Testing Design. *Pace University*, pp.1–268.

FINE, M.R., 2002. *Beta Testing for Better Software*, Wiley.

FLYSPRAY, 2012. Flyspray. Retrieved September 2, 2013 from http://flyspray.org/.

GAHRAN, A., 2011. What's a mobile app? Retrieved September 2, 2013 from http://www.contentious.com/2011/03/02/whats-a-mobile-app/.

HP, 2012. HP UFT Mobile accelerates automated mobile testing.

KIRUBAKARAN, B. AND KARTHIKEYANI, V., 2013. Mobile application testing — Challenges and solution approach through automation. *2013 International Conference on Pattern Recognition, Informatics and Mobile Engineering*, pp.79–84.

LONG, X., 2010. Adaptive random testing of mobile application. *2010 2nd International Conference on Computer Engineering and Technology*, pp.297–301.

MAHMOUD, Q.H. AND POPOWICZ, P., 2010. A Mobile Application Development Approach to Teaching Introductory Programming. *Frontiers in Education Conference (FIE), 2010 IEEE*, pp.1–6.

MOBITHINKING.COM, 2013. Mobile applications: native v Web apps – what are the pros and cons? Retrieved September 2, 2013 from http://mobithinking.com/native-or-web-app.

MUCCINI, H., FRANCESCO, A. DI AND ESPOSITO, P., 2012. Software Testing of Mobile Applications: Challenges and Future Research Directions. *Automation of Software Test (AST), 2012 7th International Workshop on*, pp.29–35.

NIELSEN, J., 2000. Why You Only Need to Test with 5 Users. Retrieved September 2, 2013 from http://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/.

NIELSEN, J. AND LANDAUER, T.K., 1993. A mathematical model of the finding of usability problems. *Proceedings of ACM INTERCHI'93 Conference*, pp.206–213.

PAN, J., 1999. Software Testing. Retrieved September 2, 2013 from http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/.

PRADHAN, T., 2012. All Types of Software Testing. Retrieved September 2, 2013 from http://www.softwaretestingsoftware.com/all-types-of-software-testing/.

RUNESON, P. AND HÖST, M., 2008. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2), pp.131–164.

SUMMERFIELD, J., 2013. Mobile Website vs. Mobile App. Retrieved September 2, 2013 from http://www.hswsolutions.com/services/mobile-web-development/mobile-website-vs-apps/.

Two-dimensional Extensibility of SSQSA Framework

JOZEF KOLEK, GORDANA RAKIĆ AND MILOŠ SAVIĆ, University of Novi Sad

The motivation to improve systematic application of software analysis tools by improving characteristics of software analysis tools originates from important aspects of modern software development regarding complexity and heterogeneity; importance of analysis and control during this process; need to keep consistency of the followed results. During the identification of the factors affecting these process we identified two important characteristics of supporting tools: extensibility and adaptability. In this paper describe extensibility of the Set of Software Quality Static Analyzers (SSQSA) in two directions: to support new programming language and to support new analysis algorithm

Categories and Subject Descriptors: D.2.8 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms: Languages, Measurement

Additional Key Words and Phrases: SSQSA, eCST representation, eCSTGenerator

1. INTRODUCTION

Nowadays, large software projects are very complex. They consist of many components, usually very heterogonous ones and developed by usage of many different programming languages and many different programming paradigms. Therefore it is very useful to have some unique set of tools that enables consistent analysis, measurement and control during software development. It is important to support large set of programming languages with different programming paradigms and to provide extensibility and adaptability of the tools. Since almost every single project is unique and every individual or group working on a given project has its own specific needs, it is very important to have flexible set of tools. By flexible we mean that it should be easily extensible and easily adaptive to current needs. One of the main weaknesses of available tools in this field is strong dependency of applicability of software metrics on input programming language [Rakić and Budimac 2011c].

Furthermore, if we consider usage of several language-specific or paradigm-specific tools in development of a single project we meet another difficulty: inconsistency of the gained results. Namely, researches [Novak and Rakić 2011, Lincke et al. 2008] show that different tools ran on the same project may produce different results.

These important aspects of analysis modern software development are motivation to improve systematic application of software analysis tools [Rakić and Budimac 2011c]. by improving characteristics of software analysis tools [Budimac et al. 2012]. A language independent intermediate representation is introduced [Rakić and Budimac 2011a]. It is basis for development of tools to support consistent analysis during software development. These tools have some common characteristics inherited from the joint internal representation. These are language independency, extensibility, and adaptability.

In this paper we will briefly describe extensible Set of Software Quality Static Analyzers (SSQSA) - Section 2. Section 3 provides another side background by describing ANTLR – tool used to make extensibility stronger. In Section 4 we will demonstrate extensibility of the set of the tools on two levels. On lower level we provide process for introducing support of a new programming language which is provided in Section 4.1. On higher level we describe how to introduce new analysis as a new functionality to the set of the tools. This is provided by Section 4.2. Section 5 demonstrates how these processes work on real examples. For these purposes we add support for Delphi (Section 5.1) and calculation of Halstead metrics (Section 5.2.). Related work is provided by Section 6. Finally, conclusion and future work are given by Section 7.

Author's address: J. Kolek, G. Rakić, M. Savić, Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia; email: jkolek@gmail.com, {goca, svc}@dmi.uns.ac.rs

In: Z. Budimac (ed.): Proceedings of the 2nd Workshop of Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA), Novi Sad, Serbia, 15.-17.9.2013, published at http://ceur-ws.org

2. SSQSA

Set of Software Quality Static Analyzers (SSQSA) [Budimac et al. 2012] is a set of software tools for static analysis of programs. It is intended to be programming language independent: to support as many languages as possible, and (more importantly) to be easily extensible in this direction. Many different, not only programming, languages are supported at the moment, e.g. Java, C#, Modula-2, WSL, OWL, etc. Program sources of different programming languages are translated into unique intermediate structure named Enriched Concrete Syntax Tree (eCST), and this is what makes it language independent.

The eCST is a new type of syntax tree to be used as intermediate representation of the source code. This intermediate representation is suitable for implementation of many different algorithms for source code analysis. eCST contains unified set of universal nodes to mark different language elements. For example we have universal nodes for marking function calls, variable declarations, operators, etc. Therefore, already existing universal nodes are sufficient to implement a wide range of algorithms for static program analysis [Rakić and Budimac 2011a].

Current architecture (Figure 1.) of the SSQSA is provided by [Rakić et al. 2013] eCST is generated by the central component of the SSQSA framework called eCSTGenerator. So far SSQSA consists of three fully functional tools:

- SMIILE - Software Metrics Independent of Input LanguagE [Rakić and Budimac 2011b],
- SNEIPL - Software Networks Extractor Independent of Programming Language [Savić et al. 2012]
- SSCA - Software Structure Change Analyser [Gerlec et al. 2012]

Other tools are in the development phase. Development of new programming language support and new tools should be straightforward. We describe these procedures in the following sections.

3. ANTLR

ANother Tool for Language Recognition (ANTLR) [Parr 2007] parser generator is the primary tool for adding and maintaining language supports. It takes the grammar specification of the language and produces scanner and parser written in different target languages.

ANTLR[i] is externally produced powerful tool that can process various types of files into syntax trees.

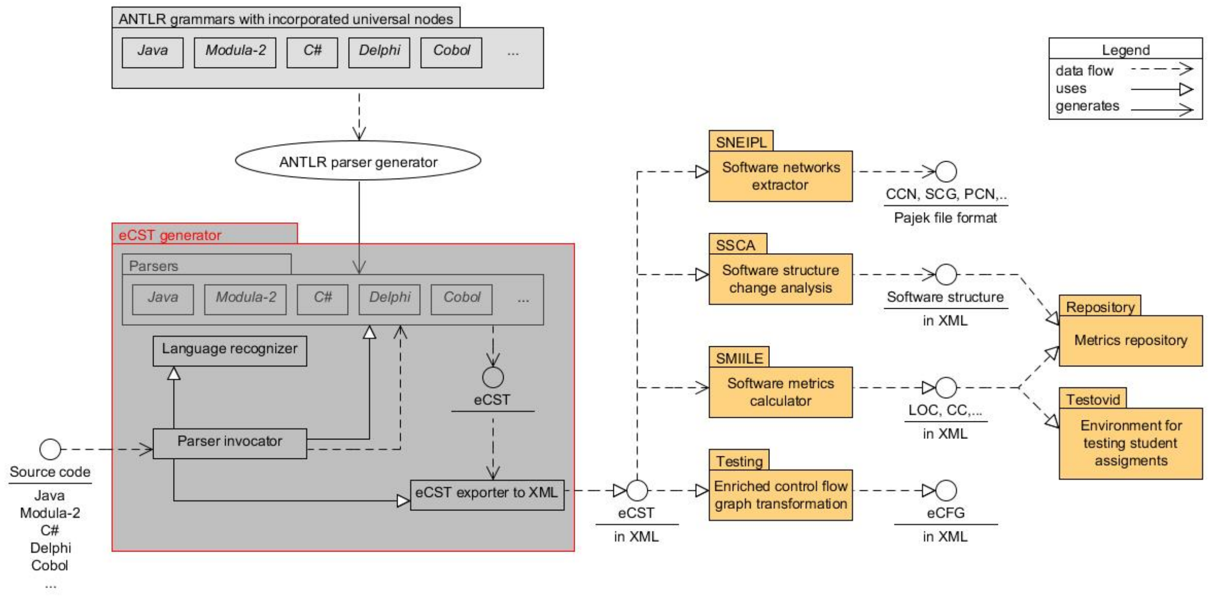


Figure 1. SQSSA architecture

One very important aspect of ANTLR parser generator tool is its ability to rewrite the grammar rules.

[i] ANTLR project site http://www.antlr.org/

This is the way the universal nodes are incorporated into generated syntax trees. This is done on a declarative level without coding in the target language. In the Section 4.1. and its subsections we provide step by step example.

However ANTLR is LL(*) parser generator, the number of look-ahead tokens can vary from rule to rule. Also, ANTLR has the very powerful feature based on backtrack algorithm to match specified rules. But this backtrack feature should be avoided when it is possible because it can be very memory- and time-consuming, and ANTLR grammar becomes hard to debug.

4. ADDING NEW FUNCTIONALITIES

In this Section we will describe general steps necessary to be made to add two new facilities to SSQSA architecture. First one is introducing support for new input language. The second one is new functionality of the tools using eCST as internal representation. This can be applied independently of whether we want to add new analysis to the one of operational tools incorporated in SSQSA or we want to add completely new tool as a component in our framework.

4.1 Introducing Support for New Input Language

To add support for new input language we will primarily need language specification. The best option is to find formal language specification by language grammar. Since ANTLR parser generator is the primary tool for adding and maintaining language supports, the most suitable form of the initial specification of new language is EBNF (Extended Backup-Naur Form) notation[i]. It is much easier to rewrite (rewrite in sense of writing generic ANTLR grammar with respect to the language definition) and modify this kind of specification in ANTLR notation. Counterpart example is language specification in BNF notation, where left recursion is natural feature and repetitions are expressed with recursion. This is kind of grammar that ANTLR notation cannot handle and hence one who introduce new language has to work harder on translation.

Finding the most ideal language specification in EBNF and with LL(1) property is very rare case, because most popular languages have ambiguous grammars. Still, any language specification can be used. When we have it than the process for introducing the language in the framework consists of the following steps:

(1) translate the given language specification to grammar in ANTLR notation (write the ANTLR grammar),
(2) add a rule for syntax tree generation to the grammar (rewrite the rules),
(3) add the universal nodes to the syntax tree (extend the rules),
(4) generate the parser and the scanner
(5) add language to the XML configuration file for supporting languages

We will describe each of these steps in detail.

4.1.1. Write the ANTLR Grammar

Structure of the ANTLR grammar and rule syntax are described on wiki pages on the ANTLR project site. Characteristic of rule syntax is that it is comparable to EBNF notation. Table 1 provides the parallel preview of the main part of the rule syntax in EBNF and in the ANTLR notation.

Table 1. Preview of EBNF and ANTLR notation

EBNF	*ANTLR*	**meaning**
`A	B`	`A	B`	alternative (A or B)
`{A}`	`A*`	zero or more repetition of A
`A{A}`	`A+`	One or more repetition of A
`[A]`	`A?`	One or zero appearance of A

[i] The International standard (ISO 14977) definition of the EBNF http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip

We provide example of simple rule representing while statement.

Rule for *while* statement in ANTLR notation

```
whileStatement : WHILE expression DO statement;
```

4.1.2. Rewrite the Rules

When generic ANTLR grammar is completed, then rewriting of rules is the next step. Rule rewriting is a technique of changing the output structure of existing rules. This is the way to create Syntax Trees. This tree is Abstract Syntax Tree (AST) in terms of ANTLR community, but in our case they are actually Concrete Syntax Trees (CST) because we do not omit any elements of the source code and all source code elements are preserved. So, the important issue in our case is that all of the syntax elements must be kept.

We extend simple rule example provided above by adding syntax tree generation

Rule for *while* statement in ANTLR notation with syntax tree generation

```
whileStatement : WHILE expression DO statement
                -> ^( WHILE expression ^(DO statement) ) ;
```

4.1.3. Extend the Rules

When rewriting of rules is done, the universal nodes can be added. This is actually the conversion from syntax tree to the enriched Concrete Syntax Tree (eCST). Let us demonstrate this step on our simple example.

Rule for *while* statement in ANTLR notation with eCST generation

```
whileStatement : WHILE expression DO statement
                -> ^( LOOP_STATEMENT
                        ^( KEYWORD WHILE )
                        ^( CONDITION ^( EXPR expression ) )
                        ^( KEYWORD DO )
                        statement);
```

So far catalog of universal nodes consists of more than 30 nodes. For example we have universal nodes to describe function and procedure calls, variable declarations, branch and loop statements, etc. Previous version of catalog is available at [Gerlec et al., 2012]

4.1.4. Generate the Parser and the Scanner

This step is very straightforward by running ANTLR parser generator. It can be done manually via console or it can be done automatically with help of some integrated development environment.

4.1.5. Add a XML Support for the Language

SSQSA eCSTGenerator dynamically recognizes input language in the input file based on the extension of the input file. It calls appropriate scanner and parser and generates eCST. For this purposes we store all needed information about supported languages to the XML file. It contains all information needed to recognize language, call scanner and parser and generate the tree without interaction with the user. XML Schema for storing configuration data about supported input languages is provided by Figure 2.

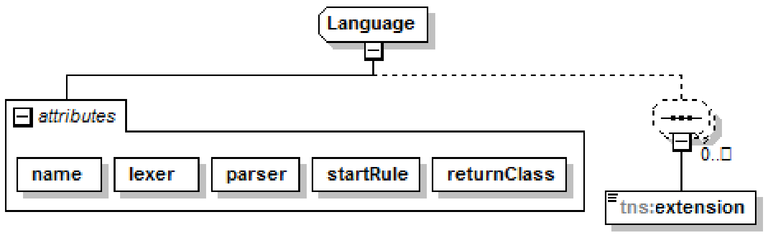


Figure 2. XML Schema for storing configuration data about supported input languages

Finally, eCST representation of the source code is saved in the XML file as well, and it is ready to be used by available tools. For every compilation unit one XML file is created.

4.2 Adding an Analysis

Let us assume that we already have some set of input languages supported by SSQSA environment. Let us consider what steps are needed to implement new functionality on these languages.
To accomplish this task we need to follow this procedure:

(1) define the set of universal nodes needed to implement wanted algorithm

(2) if necessary add the new nodes to existing ANTLR grammar as it is described before (for all languages)

(3) if grammar has been modified generate the scanner and the parser (for all languages)

(4) traverse the eCST, and use the incorporated universal nodes to accomplish the analysis.

4.2.1. Determine the Set of Needed Universal Nodes

The very first step in introducing new functionality is to analyze the algorithm to be implemented. This should lead to determining the set of nodes we need to implement the algorithm. During this analysis we have to think of existing nodes in the catalog. Often existing nodes can be reused and this is very desirable to do so, because we do not want to have two or more different nodes for similar or identical purpose. Our goal is to keep set of universal nodes as minimal as possible and to cover our needs as much as possible. However, if the set of existing nodes does not fit into our requirements, then addition of the new nodes should be considered.

4.2.2. Extend the Existing ANTLR Grammars

Conditionally, in the case when existing universal nodes do not satisfy our needs the new nodes need to be added. This step is already explained in Section 3.1.4. It is very important to do this for all supported languages to keep consistency and completeness. The other important note is to save the nodes previously introduced because we need them for existing implementations.

4.2.3. Generate Scanners and Parsers

Generation of the Scanner and the Parser can be done as explained in 3.1.5. After generation of scanner and parser, they can be used to parse input files and generate eCST also for the already existing functionality because previously used nodes haven-t been modified. However, this step is needed only in case when the grammar file has been modified. In that case we will regenerate scanner and parser for all modified grammars.

4.2.4. Implement the Analysis Algorithm

After an eCST is generated by eCSTGenerator, we can implement an algorithm that traverses the tree, collects the information and does wanted analysis.

5. EXAMPLE

In this Section we will demonstrate described extensibility on the example. We introduce new language (Section 4.1.) and implementation of new software metric calculation (Section 4.2.).

5.1. Delphi

To demonstrate extensibility of SSQSA framework to introduce support for new input language we introduce support for Delphi. Delphi is characteristic in a way that it has elements of both, structural and object-oriented language.
As we mentioned before, first step in adding new language to SSQSA architecture is to find language specification in EBNF notation. As a starting point Delphi 6[i] language specification in EBNF notation has been used. Since Delphi language is derivation of the Object Pascal, earlier versions are very similar to this language, and therefore have LL(1) property in most of the rules, which is suitable to translate to our ANTLR notation. However, translating of given Delphi language specification to our ANTLR notation was not so straightforward, because some parts of given Delphi language specification are very complicated. At some places ANTLR backtrack option was used. During the translating to ANTLR notation Delphi grammar was constantly tested on quite large set of test cases to ensure correctness of the derived grammar.

When generic grammar in ANTLR notation was done, the next step was to rewrite the grammar rules. This step was pretty straightforward. After this step generation of Syntax Tree was supported. In the terms of ANTLR notation it is called Abstract Syntax Tree, but since we keep all of the syntax elements it was closer to Concrete Syntax Tree. After this, grammar was ready for incorporation of the universal nodes. At this step, at some places it was necessary to restructure some of the grammar rules to fit our needs. After universal nodes were incorporated, the next step was to generate the scanner and the parser, and to add needed information to the “Languages.XML” configuration file.

First run [Rakić et al. 2013] of eCSTGenerator on large Delphi project “DelphiProp” consisting of 104438 Lines of Code gave results presented in the Table 2. The number of produced eCST trees (compilation units), the total number of eCST nodes contained in produced trees, running time in seconds and the storage size needed to export eCST trees into the XML files produced by eCST Generator are provided. It can be seen that the transformation of source code into the eCST representation lasted less than a minute where produced eCST trees contains nearly of more than one million nodes.

Furthermore, Table 2 provides the results of running SNEIPL tool [Savić et al. 2012,] which is part of SSQSA environment [Rakić et al. 2013] on this project. This tool generated General Dependency Network (GDN) of the project. We provide the number of GDN nodes, the number of GDN links and the time needed to generate the network. The experiment was performed on AMD Athlon 3200+ processor with 1GB RAM memory.

Table 2. Results of running of eCSTGenertor and back/end SSQSA tools on DelphiProp project

SMIILE	***eCSTGenerator***				***SNEIPL***		
LOC	**#eCST**	**#nodes**	**T[s]**	**S[MB]**	**#GDN nodes**	**#GDN links**	**T[s]**
104438	491	1099961	31	61.7	13721	17745	81

5.2. Halstead Metrics

Halstead metrics express program size and complexity which is evaluated directly from source code. Calculation of Halstead metrics are based on number of occurrences of the operators and the operands. In SSQSA environment the calculation of Halstead metrics can be divided into two phases:

(1) parsing the source code and generation of the corresponding eCST,

(2) calculations of Halstead metric.

First phase takes Delphi source code as input, and generates the eCST, which is then input of the second phase. This is to be done by eCSTGenerator. Second phase, which is actually the core of Halstead metrics

[i] Delphi 6 starting grammar web site http://dgrok.excastle.com/Grammar.html

calculations, traverses this eCST, calculates the Halstead metrics and outputs the results. While traversing the tree, the implemented algorithm must count the total and distinct number of occurrences of the operators and the operands.
The Halstead's operators are: keywords, operators such as “+” and “-”, and separators. On the other hand Halstead's operands are: identifiers, constants, types, and directives.

To recognize keywords, operators and separators the algorithm for computing Halstead metrics uses KEYWORD, OPERATOR and SEPARATOR universal nodes, respectively. TYPE_TOKEN, DIRECTIVE and CONST universal nodes are used to identify operands. It is important to note the difference between the following universal nodes:

- ☐ TYPE marks identifiers representing user-defined data types, and
- ☐ TYPE_TOKEN marks primitive, built-in types provided by a programming language.

Universal node NAME is used to collect information about identifiers.

Initial test cases were selected in that way that generated values can be manually verified. Table 3 provides results for one of the largest test cases from this category.

Table 3. Halstead metric values for a test case

Distinct Operators (n1)	*63*	***Program Vocabulary (n)***	*164*	***Program Difficulty (D)***	**105,41584**
Distinct Operands (n2)	101	**Program Length (N)**	1081	**Programming Effort (E)**	838426,3
Total Operators (N1)	743	**Program Volume (V)**	7953,5137	**Programming Time (T)**	46579,24
Total Operands (N2)	338	**Program Level (L)**	0,00948624	**Intelligent Content (I)**	75,448944

6. RELATED WORK

Since the main feature of SSQSA system is its language independency of its internal representation of the source code and therefore extensibility in that direction, we concentrate mainly on this feature in analysis of similar achievements.
Following our overall goal we come to only one related research and development project[i] [Bär 1999]. FAMIX - family of meta-models [Lanza, and Marinescu, 2006] and MOOSE – an extensive platform for software and data analysis [Ducasse et al., 2000] have the most similar general goals to our project. Their strength is mainly in language independency. They support OO design (at the interface level of abstraction) for a wide range of input programming languages. Different input languages are supported by separate tools so-called importers for filling in the meta-model with the information from the source code. This means that it is needed to implement importer tool for each new language which is to be supported. By usage of eCSTGenerator and eCST representation of the source code we enable user just to prepare appropriate grammar to get the full support of needed language. We believe that our approach is more general and more flexible. eCST used to represent the source code covers all aspects of source code and not only the design. It is thus equally appropriate to support broader set of static analysis algorithms. However, it also fully supports procedural languages, including the legacy ones (e.g., COBOL), but also Domain Specific Languages such is OWL and WSL.

We can also discuss situation in the domains that our corresponding back-end tools cover, e.g. software metrics and network extraction.

Similar approach to ours was detected in the ATHENA project [Christodoulakis et al., 1989]. It was tool for assessing the quality of software and the final goal of the tool was to generate a report that describes the quality. ATHENA was based on the parsers that generate abstract syntax trees as a representation of a source code. Used parsers were manually implemented for each language to be supported and algorithms for calculation of software metrics were partially inbuilt in parser implementation. The generated trees were structured in such a way that the metric algorithms were

[i] FAMOOSE project web site http://scg.unibe.ch/archive/famoos/

easily applied. This is the week point regarding extensibility if we have in mind that for each new language one have to develop new parser with inbuilt metric algorithms in opposite to our approach to generate parsers by parser generator in order to automate process of adding support for new language. Furthermore, eCST is richer representation then AST. Finally, ATHENA was only executable under the UNIX operating system and its official support is not available anymore. SSQSA framework and SMIILE tool, its equivalent to ATHENA (by purpose) is implemented in Java and therefore it can be used on broader range of platforms.

Another tool with similar approach is the CodeSquale[i] metrics. This project was based on a similar idea and the same final goal - language independency. The authors developed a system based on the representation of a source code by AST and implemented one object-oriented metric for the Java source code. Furthermore, an idea for the additional implementation of other metrics and opportunities for extending the tool to other programming languages was described. Unfortunately, later results were not published. However, week point of this project was usage of AST for representing the source code. By using eCST we get broader set of algorithms implementable independently of programming language.

Let us look at the field of software networks extraction tools. There is a variety of software networks extractors, but in most cases their usage is restricted to a particular programming language and extract just one type of software network (for example, review of static call graphs extractors for C programming language can be found in [Murphy et al 1998]). It can be concluded that there is no a language independent tool for extraction of software networks covering different levels of abstractions. Furthermore, no tool has possibility to introduce support for new language. If we have in mind limitation concerning range of covered languages and types of supported networks we consider that no available tool for network extraction can meet characteristics that SNEIPL posses and that is extensibility and wide application.

7. CONCLUSION AND FUTURE WORK

In this paper we described SSQSA environment and step by step demonstrated its extendibility. This was also represented by appropriate examples. The approach by usage of eCST applied in SSQSA is very flexible and extensible. Furthermore, eCST representation of the source code is suitable to implement a wide range of algorithms for static code analysis. This means that eCST can be used in some other projects with minor or none modifications. Finally, it can be extended with totally new nodes to satisfy various needs. However, characteristics of SSQSA environment can gain additional value by engaging more automated processes in the whole idea.

Since there are many different grammar notations, both attributed grammars for corresponding parser generators and generic EBNF notations that serves as documentation of some programming languages, it would be useful to have some kind of automatic translators from other grammar notations to ANTLR notation and vise versa. This idea introduces many new questions to the subject, for example how to translate notations with left recursion allowed to notation with LL(*) property (like the ANTLR notation is), e.g. how to deal with actions in attributed grammar notations.

Also, the usage of alternative parser generator can considered to make it easier to find grammar and generate parser. Sometimes it is easy to find grammar for the particular language, but its translation to the ANTLR notation requires hardworking. The previous idea for automated translation between notations of the different parser generators can also be incorporate here.

Moreover, introducing the tool that enables visual creation of the grammar could be more than helpful. If it would be possible to generate parser by only drawing the syntax diagrams or editing the rules in some alternative visual representation this would add important advantage to our approach.

REFERENCES

H Bär. The FAMOOS Object Oriented Reengineering Handbook. Edited by Stéphane Ducasse. Forschungszentrum Informatik an der Univ., 1999.

Zoran Budimac, Gordana Rakić, Marjan Heričko, Črt. Gerlec.. Towards the Better Software Metrics Tool, In Proc of 16th European

[i] CodeSquale project web site http://code.google.com/p/codesquale/

Conference on Software Maintenance and Reengineering (CSMR), Szeged, Hungary, March 27-30, 2012, pp. 491-494.

Zoran Budimac, Gordana Rakić, Miloš Savić, SSQSA architecture, In Proc. Of the Fifth Balkan Conference in Informatics (BCI 2012), Novi Sad, Serbia, September 16-20 2012, ISBN: 978-1-4503-1240-0 doi: 10.1145/2371316.2371380, pp. 287-290

D. Christodoulakis, C. Tsalidis, C. van Gogh, V. Stinesen, Towards an automated tool for software certification. In Proc of Tools for Artificial Intelligence, 1989., IEEE International Workshop on Architectures, Languages and Algorithms. pp. 670–676.

S Ducasse, M Lanza, S. Tichelaar, Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In Proc. of 2nd International Symposium on Constructing Software Engineering Tools (CoSET), Limerick Ireland, 4 – 11 June 2000

Črt Gerlec, Gordana Rakić, Zoran Budimac, Marjan Heričko, 2012. A programming language independent framework for metrics-based software evolution and analysis. ComSIS journal, Vol. 9, No. 3, 1155-1186. (2012).

M. Lanza, and R. Marinescu. Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Springer, 2006

R Lincke, J Lundberg, W Löwe, Comparing software metrics tools. In Proc. of the international symposium on Software testing and analysis(ISSTA '08). ACM, New York, NY, USA, 131-142. (2008)

Gail C. Murphy, David Notkin, William G. Griswold and Erica S. Lan. 1998 An empirical study of static call graphs extractors. ACM Transactions on Software Engineering and Methodology. 7, 2 (April 1998) 158-191

Jernej Novak, Gordana Rakić, Comparison of Software Metrics Tools for :NET, In Proc. Of 13th International Multiconference Information Society (IS), Collaboration, Software And Services In Information Society (CSS), October 11-15, 2011, Ljubljana, Slovenia, vol. A, pp. 231-234

T. Parr, The Definitive ANTLR Reference - Building Domain-Specific Languages, The Pragmatic Bookshelf, USA, 2007, ISBN: 0-9787392-5-6.

Gordana Rakić, Zoran Budimac, 2011a. Introducing Enriched Concrete Syntax Trees, In Proc. of the 14th International Multiconference on Information Society (IS), Collaboration, Software And Services In Information Society (CSS), (Ljubljana, Slovenia, October 10-14), Volume A, pp. 211-214,

Gordana Rakić, Zoran Budimac, SMIILE Prototype, 2011b. In Proc. Of International Conference of Numerical Analysis and Applied Mathematics ICNAAM2011, Symposium on Computer Languages, Implementations and Tools (SCLIT), (Halkidiki, Greece, September 19-25, 2011) ISBN 978-0-7354-0956-9, pp. 853-856.

Gordana Rakić, Zoran Budimac, 2011c. Problems In Systematic Application Of Software Metrics And Possible Solution, In Proc. of The 5th International Conference on Information Technology (ICIT) (Amman, Jordan, May 11-13, 2011).

Gordana Rakić, Zoran Budimac, Miloš Savić, Language Independent Framework for Static Code Analysis, In Proc. Of the Sixth Balkan Conference in Informatics (BCI 2013), Thessaloniki, Greece, September 19-21 2013, in print

Miloš Savić, Gordana Rakić, Zoran Budimac and Mirjana Ivanović, Extractor of Software Networks from Enriched Concrete Syntax Trees, In Proc. Of International Conference of Numerical Analysis and Applied Mathematics ICNAAM2012, 2nd symposium on Computer Languages, Implementation and Tools (SCLIT), AIP Conference Proceedings, Sep. 2012, vol. 1479, pp. 486–490

Towards New User Interfaces Based on Gesture and Sound Identification

KRISTJAN KOŠIČ, BOŠTJAN ARZENŠEK AND SAŠA KUHAR, University of Maribor
MATEJ VOGRINČIČ, University Medical Center Maribor

The relatively low price of devices that enable capture of 3D data such as Microsoft Kinect will certainly accelerate the development and popularization of a new generation of user interaction in the business application domain. Although the application interfaces and libraries that make it easier to communicate with these devices continue to be in the process of developing and maturing, they can still be used for the development of business solutions. In addition, gestures and sounds provide more natural and effective ways of human-computer interaction. In this paper we present an overview and a basic comparison of the available sensing devices together with the experience gained during the development of solution ADORA, which main purpose is to assist surgeons with the help of contactless interaction.

Categories and Subject Descriptors: H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Evaluation/ methodology*

General Terms: Natural user interfaces,Human Computer Interaction

Additional Key Words and Phrases: depth vision, natural user interfaces, healthcare, sensors, kinect

1. INTRODUCTION

The average user communicates with the computer using a keyboard and a computer mouse. The keyboard remains at the core computer interaction since the first commercial computer in 1984. In more than half a century since the invention of the first computer mouse many pointing devices have been introduced, of which best known are a tracking mouse (trackball) and light pen. None of these devices worked out to be better than the keyboard, so the majority of human-computer interaction (HCI) is performed via computer keyboard and mouse, which are still the same as they were at the time of the invention. With the rapid advances of technology other ways of HCI were developed. In the last decade, a noticeable use of touch screen devices and other innovative gaming interfaces were developed and successfully used in practice. At the same time with technology development new challenges emerged, e.g. "How to communicate with computers using complex commands without direct physical contact?" The solution would facilitate and optimize work in many specialized domains. Alexander Shpunt [Dibbell 2011] introduced three-dimensional (3D) computer vision. Simple communication and control of a computer by using the user's movements (gestures) and voice commands was enabled. The sensing device records observed space, takes an image and converts it into a synchronized data stream. Data stream consists of depth data (3D vision) and color data (similar to human vision). Depth vision technology was invented in 2005 by Alexander Shpunt, Zeev Zalevsky, Aviad Maizels and Javier

Author's address: K. Košič, B. Arzenšek, S. Kuhar, Institute of informatics, Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova ulica 17, 2000 Maribor, Slovenia, email: kristjan.kosic@um.si, {bostjan.arzensek, sasa.kuhar}@uni-mb.si; M. Vogrinčič, University Medical Center Maribor, Ljubljanska ulica 5, 2000 Maribor, Slovenia, email: matej.vogrincic@ukc-mb.si

In: Z. Budimac (ed.): Proceedings of the 2nd Workshop of Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA), Novi Sad, Serbia, 15.-17.9.2013, published at http://ceur-ws.org

Garcia [Zalevsky et al. 2007]. The technology has been established in the world of consumer technology (gaming consoles). Massachusetts Institute of Technology (MIT) ranked gesture interface based technology among the top ten most successful technologies in 2011 [Dibbell 2011]. The major console manufacturers (Microsoft, Sony and Nintendo) upgraded their gaming experience with an advanced motion-sensing interfaces. Sony and Nintendo have developed a wireless controllers (PlayStation Move and Wii MotionPlus), while Microsoft's Xbox console used a completely noncontact approach with the new Kinect sensor. Microsoft Kinect sensor is currently the most visible and easily accessible gaming controller on the market. Microsoft's decision to publish development libraries Kinect for Windows SDK, enabled rapid growth and paved the way for a wide range of potential applications in different domains (medicine, robotics, interactive whiteboards, etc.). Microsoft Kinect is the first commercial composite interface that combines camera to detect body movements and facial and voice recognition. Gartner's hype cycle for HCI technologies [Prentice and Ghubril 2012] mentioned that gesture interface control technology is steadily moving towards greater productivity, a mature market and higher returns of value. Company analytics at Markets & Markets [Marketsandmarkets.com 2013] have evaluated the market value of the hardware and software that allows you to control your computer using gestures and voice commands, such as the Microsoft Kinect, to U.S. $ 200 million in 2010. According to recent market research data, the value of contactless interaction and gesture recognition in 2018 will reach 15 billion U.S. dollars [Marketsandmarkets.com 2013]. HCI defines user experience. Gesture interface control enables the recognition and understanding of human body movement for the purpose of interaction and control of computer systems without direct physical contact [Prentice and Ghubril 2012]. The term "natural user interface" is used to describe systems that communicate with the computer, without any intermediate devices. In the last decade a big leap forward was made from the traditional ways of managing computer software with keyboard and mouse, to non-contact HCI, which was primarily used in the gaming field.

2. SENSING DEVICES FOR HCI

Three types of sensing technologies mainly occur in computer vision research: stereo cameras, time-of-flight (ToF) cameras and structured light [Chen et al. 2013]. Stereo machine vision is biomimetic, where 3D structure is gathered from different viewpoints, similar to human vision. Time of flight cameras estimate distance to an object with the help of light pulses from a single camera. Information such as time to reach the object and speed of light define the distance from the measured point. ToF devices have high precision and are very expensive. Structured light sensors started to develop with the help of PrimeSense technology, which was acquired by Microsoft and built into their Kinect sensor. Main advantage of structured light sensors is the price-performance balance. Microsoft Kinect is priced at consumer level and still obtains sufficient precision level [Gonzalez-Jorge et al. 2013].

2.1 Depth sensing

Vision sensing devices capture distance from the real world, which cannot be obtained directly from an image. Depth images require pre and post-processing. Depth view enables body part detection, pose estimation and action recognition. Body parts and pose detection is a popular topic, while action recognition is starting to receive more research attention [Chen et al. 2013]. A lot of research [Clark et al. 2012; Dutta 2012; Gabel et al. 2012; Khoshelham and Elberink 2012; Stoyanov et al. 2012] is done in the field of body data detection and its transformation to active skeletons. There are still some precision issues as mentioned by [Khoshelham and Elberink 2012], but if there is no need for high precision tracking, currently available devices handle the issues quite successfully.

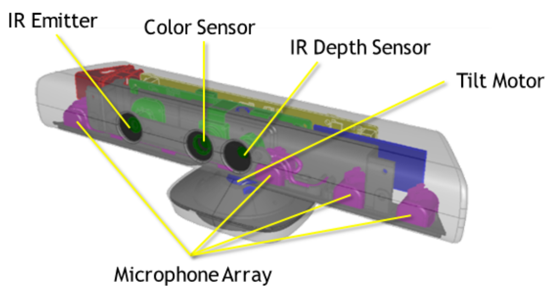


Fig. 1. Kinect sensor structure [Microsoft 2013].

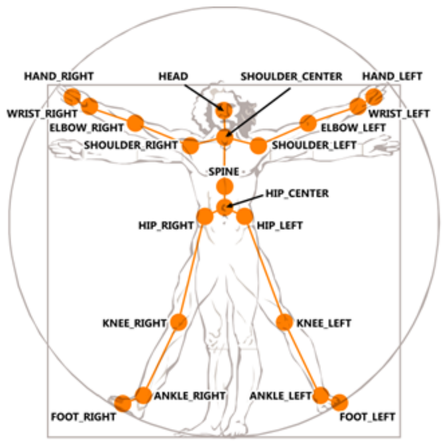


Fig. 2. Skeleton detection points [Microsoft 2013].

2.2 Microsoft Kinect sensor family

Microsoft first announced the Kinect sensor in 2009 under the name ”Project Natal”. In 2012 Kinect for Windows sensor was announced, that enabled use of advanced gesture based functionality in the business application domain. The sensor was accompanied by software development library – Kinect for Windows SDK. Until February 2013 Microsoft sold 24 million units of the Kinect sensor. In the first sixty days on the market eight million units of the Kinect sensor were sold, which gave Kinect the title of fastest selling consumer electronic device and was recorded in the Guinness Book of Records[1]. The sensor has the ability to capture audio, color video and depth data (figure 1). Depth data is detected with the use of infrared light, with which a covered skeleton (figure 2) of a tracking person is formed. Kinect sensor captures depth image by using two separate data streams. The first stream presents the data from the Kinect sensor to the nearest object in millimeters and the second one presents the segmented data from a tracked person. Deep video supports the default resolution of 640x480 pixels, the value can be set to 320x240 or 80x60 pixels. Sensor can recognize up to six different people. According to [Dutta 2012] Kinect sensor was able to capture relative 3D coordinates of markers of 0.0065 m, 0.0109 m, 0.0057 m in the x, y, and z directions. Tests provided such accuracy over the range from 1.0 m to 3.0 m. This means Kinect provides very accurate data in defined ranges.

2.3 Sensors from PrimeSense family

PrimeSense sensor family exists on the HCI market from the very beginning. Inventors [Zalevsky et al. 2007] of depth vision founded their own company called PrimeSense. Their technology expanded with mass production of the Kinect sensor. PrimeSense sensor family consists of Carmine and Capri sensors. Capri 3D sensor is an embedded sensor, which main advantage is in its small size. Sensor aims at the market of mobile devices (mobile phones, tablet computers, smart televisions). Despite its small size it has great potential. At this year’s conference Google IO 2013 a prototype tablet integration with Capri 3D sensor was presented [Crabb 2013]. All sensors are accompanied with software developer kits in the form of open source libraries like OpenNi supported by NITE algorithms.

[1]Guiness World Records http://www.guinessworldrecords.com.

2.4 Asus sensor Xtion

Xtion sensor is available in two versions, Xtion and Xtion Pro Live. It is based on the same depth vision technology as the Kinect sensor family. Xtion is promoted exclusively as a PC sensor and does not require additional power supply [Gonzalez-Jorge et al. 2013]. Asus sensors are used in the same way as Kinect, but for supporting software library OpenNi framework is used. Framework can be used in conjunction with Microsoft Kinect or other sensors based on PrimeSense technology.

2.5 Leap Motion sensor

Leap Motion sensor is a small device with enormous potential and aims to change the way we interact with computers. Sensor is installed next to the keyboard on the office desk and it provides a very accurate individual fingers detection. It is much more accurate than Kinect sensor (up to 200 times) [Hodson 2013]. This makes user interaction very precise and domain specific. The main purpose of Leap Motion sensor is not the depth vision of 3D space, but the exact finger detection and integration of these functionalities with existing applications

2.6 MYO Sensor

MYO is a not a depth vision sensing device but an intelligent armband. It detects motion in two ways: muscle activity and motion sensing [Nuwer 2013]. The MYO uses Bluetooth 4.0 Low Energy to communicate with the paired devices. It features on-board, rechargeable Lithium-Ion batteries and an ARM processor. Sensor is outfitted with proprietary muscle activity sensors. It also features a 9-axis inertial measurement unit. Muscle activity is defined by measuring electric activity in muscles, known as electromyography (EMG). Table I shows a basic comparison between the above-mentioned and other currently available devices.

Table I. Sensing device comparison

DEVICE	CONTROL	DATA SOURCE	VIDEO RESOLUTION	HCI
KINECT XBOX 360	contact-free	A/V/IR	640 x 480 30fps, 320X240 30fps	GS,VC
KINECT FOR WINDOWS	contact-free	A/V/IR	640 x 480 30fps, 320X240 30fps	GS,VC
KINECT ONE	contact-free	A/V/IR	1920 x 1080 60fps	GS,VC
ASUS XTION	contact-free	A/V/IR	1280 x 1200 30fps, 60fps	GS,VC
PRIMESENSE CAPRI	contact-free	A/V/IR	640 x 480	GS
PRIMESENSE CARMINE	contact-free	A/V/IR	640 x 480 30fps	GS,VC
LEAP MOTION	contact-free	-/V/IR	/	GS
SONY MOVE	with controller	A/V/-	640X480 60fps, 320x240 120fps	GS,VC
WII MOTIONPLUS	with controller	-/-/IR	none	GS
MYO	armband	EMG	none	GS

A-audio, V-video, IR-infrared; fps – Frames per Second
GS – Gesture Support, VC – voice control
EMG – Electromyography

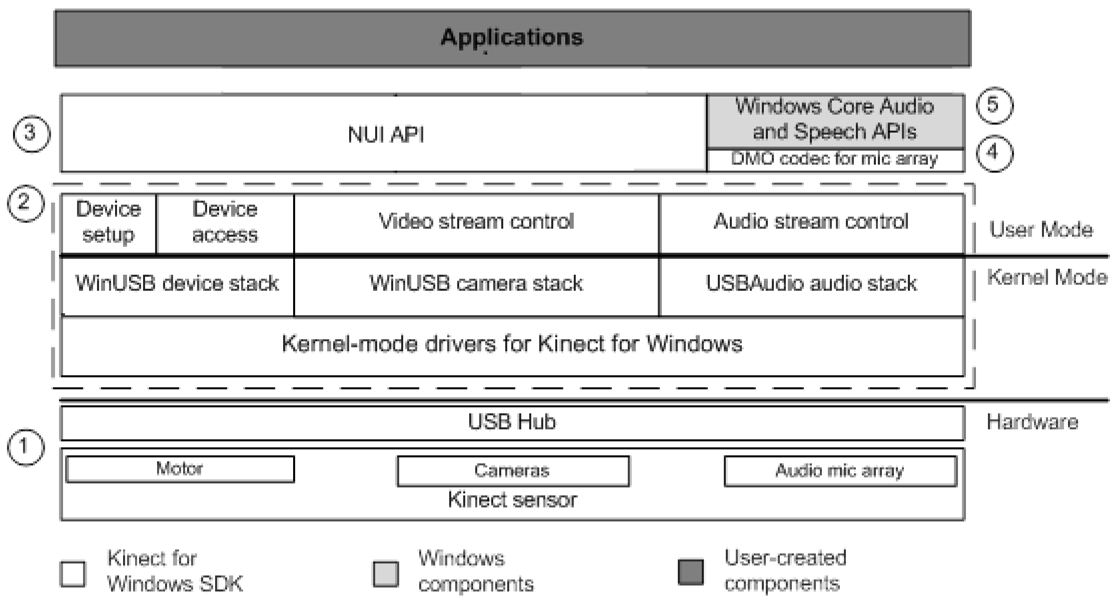


Fig. 3. Kinect SDK architecture [Microsoft 2013].

3. APPLICATION COMMUNICATION INTERFACES

The quick adoption of gesture based devices was enabled with the development of supporting software libraries (SDK). Libraries enable development of new innovative solutions and provide an upgrade to existing applications. Several libraries are publicly available. OpenKinect is an open community that uses Kinect sensor for research. The framework has a very active community, which contributes with a large set of libraries and plug-ins, thereby extending the OpenNi framework. The OpenNi core consists of several components that take care of: (i) analysis of the entire body, (ii) analysis of individual items (finger detection), (iii) recognition of user gestures and (iv) analysis of the environment (objects and people) [OpenNI 2013].

Kinect SDK [Microsoft 2013] enables software developers to develop interactive applications with support for voice command and gestures. Figure 3 shows the architecture components of Kinect for Windows SDK. These components include the following:

(1) Hardware components, including the Kinect sensor and the USB hub
(2) Windows drivers for the Kinect, which are installed as part of the SDK The Kinect drivers support:
    (a) Kinect microphone array as a kernel-mode audio device that you can access through the standard audio APIs in Windows.
    (b) Audio and video streaming controls for streaming audio and video (color, depth, and skeleton).
    (c) Device enumeration functions that enable an application to use more than one Kinect.
(3) Audio and Video Components
(4) DirectX Media Object (DMO) for microphone array beam forming and audio source localization.
(5) Windows standard APIs - audio, speech, and media APIs in Windows as described in the SDK and Microsoft Speech SDK [Microsoft 2013].

Kinect library includes advanced components known as Microsoft.Kinect.Toolkit.Controls plugin. Toolkit controls enable realization of the advanced functionality for desktop applications.

Table II. Software libraries provided with human computer sensors

LIBRARY	CAPTURE	OPEN SOURCE	SENSOR	PLATFORM
KINECT FOR WINDOWS SDK	Yes	No	Kinect	Windows
OPENNI FRAMEWORK	Yes	Yes	PrimeSense family	Windows/Linux/MacOS
OPENKINECT FRAMEWORK	Yes	Yes	Kinect	Windows/Linux/MacOS
POINTCLOUD LIBRARY	Yes	Yes	Kinect, PrimeSense	Windows/Linux/MacOS
LEAP MOTION SDK	Yes	No	Leap Motion	Windows/Linux/MacOS
THALMIC LABS MYO SDK	No	No	MYO armband	Windows/MacOS

The Point Cloud Library (PCL) [PointCloud 2013] is a standalone, large scale, open framework for 2D/3D image and point cloud processing that enables advanced 3D data analysis. It contains numerous state-of-the art algorithms that can be used to filter outliers from noisy data, stitch 3D point clouds together, segment relevant parts of a scene, extract keypoints and compute descriptors to recognize objects. PCL library can create surfaces from point clouds and visualize them [Rusu and Cousins]. The table II below summarizes current software libraries for HCI.

4. LESSONS LEARNED DURING DEVELOPMENT OF A GESTURE BASED SOLUTION ADORA

The objective of any health care institution is to optimize the length of a surgical procedure and increase its quality. ADORA[2] is an interactive physician's assistant enabling a unique presentation of information about a patient before and during surgical procedures. ADORA offers a comprehensive and integrated natural user interface experience for physicians. It is a product of field knowledge, modern information and communication technologies as well as advanced hardware. With its simple use of contact-free interaction it shortens the duration of surgeries and indirectly affects the environmental and economic aspects of healthcare. By using ADORA, physicians are able to actively participate in a surgery also outside the operating theatre. Modern methods of HCI (gestures and voice support) have been integrated into ADORA solution during development. Physicians gained control over patient data with the help of contact-free interaction. Lessons learned and best practices are summarized below and consist of following challenges:

(1) The design of graphical natural user interfaces adapted to support gestures and sound.
(2) Calibration of the sensor and the correct choice of the active detected person.
(3) Proper detection and identification of the sound source.
(4) The correct interpretation of voice command.
(5) Implementation of advanced gestures and functionality that are not supported in the basic development library Kinect (point based rotation, dynamic zoom with feedback, traceable and flexible display of DICOM images [König 2005]).

The first challenge was to design an intuitive and adaptive graphical interface that supported communication with the Kinect sensor. It is very important to include clear feedback to the user (either graphical or voice), especially when designing interactive applications. It is also recommended to include interactive help where the user has the ability to practice applications gestures and voice commands. Interactive help enables a user to learn which gestures and sound commands are required

[2]Advanced Doctor's Operational Research Assistant, http://www.adora-med.com.

to control desired functionalities. During the graphical user interface design, we had to move away from the ”classic” design of user interfaces and address the challenges associated with new ways of interaction that is typical for gesture based applications. Analysis of existing gesture based solutions and Kinect user interface design guidelines helped us in the process of natural user interface design. The user interface had to be adapted to the new user controls. Another challenge was Kinect sensor calibration. Problems arose during user detection from a variety of distances from the sensor and the detection of primary user in the presence of other surgeons. In the operating room there are at least three or more people standing close to each other. Kinect Sensor detects all users as active bodies (skeletons). Kinects correction factors helped us to calibrate the sensor according to the scope and area of usage. Detection and tracing of primary user was solved with the use of voice commands. Kinect sensor detects sound source area, which allowed us to locate the primary person in the room. Primary user is selected by executing a voice command “Follow me”. The user who executed the command becomes active person and a tracked skeleton. Detection and understanding of voice commands represented a special challenge that required knowledge of foreign languages and perfect pronunciation of these. Kinect language support is limited to thirteen languages. Recognition of voice commands takes place in stages. First, the sensor compares the sounds with the selected grammar. Grammar can be determined dynamically or is defined statically in the form of an XML document. Synonyms for each voice commands can also be defined. For example, the voice command “next” can have synonyms such as “forward” and “continue”. Detection of voice command returns sensor detection level called “confidence level ratio”. The ratio has a value from zero to one, where zero means a very weak detection and the value of one very powerful detection. In order to successfully detect voice commands the proper accent, pronunciation and intonation must be satisfied. Sound commands allow navigation through the application, item selection and object manipulation. Special digital medical image format DICOM (Digital Imaging and Communications in Medicine) [Blazona and Koncar 2007] is used in healthcare. One of the challenges was the magnification functionality. In addition to vertical and horizontal axes, depth information from the Z-axis was needed. Z-axis represents the distance from between the person and the sensor. A correct zoom factor had to be calculated. The problem was solved by adapting to the speed of hand movement. This kind of image magnification proved to be very efficient, since the user is able to control the speed and direction of zooming. It was also necessary to determine the center point of the zoom. The problem was solved with coordinate transformation from the current hand position on the widget, to a pixel point on the image itself.

5. CONCLUSION

The revolution that began in the living rooms of innovative researchers continues. Technological challenges of 3D vision have been successfully addressed. Rapid growth of sensors that enable skeleton identification, finger gestures and voice command is expected, especially the rise of applications that will incorporate their functionality. Applications that will enable contactless control of computers and other devices will slowly but surely begin to replace existing ones. Devices are already changing the way we communicate with computers. The internet of things effect is becoming more and more visible. New technologies are replacing the use of devices such as a keyboard and mouse. Touchscreens are already transforming the way we interact with mobile devices. The next step is utilization of gestures and voice commands for computer interaction in our every day. There is still some work needed to be done. Accuracy and precision decrease with range and makes usage of such devices limited to special domains. Accuracy issues can be solved with alternative types of sensing, that provide a different way of gathering motion data that is not based on depth vision (MYO armband). Sensing devices will be built into monitors, laptops, televisions and mobile devices. Given the incredible development of increasingly advanced and intuitive electronic devices we can predict that in the near future systems with similar (if not better) functionalities, as seen in science fiction movies, will be used.

REFERENCES

Bojan Blazona and Miroslav Koncar. 2007. HL7 and DICOM based integration of radiology departments with healthcare enterprise information systems. *International Journal of Medical Informatics* 76, Supplement 3, 0 (2007), S425–S432. DOI:http://dx.doi.org/10.1016/j.ijmedinf.2007.05.001

Lulu Chen, Hong Wei, and James Ferryman. 2013. A survey of human motion analysis using depth imagery. *Pattern Recognition Letters* 34, 15 (2013), 1995–2006. DOI:http://dx.doi.org/10.1016/j.patrec.2013.02.006

Ross A. Clark, Yong-Hao Pua, Karine Fortin, Callan Ritchie, Kate E. Webster, Linda Denehy, and Adam L. Bryant. 2012. Validity of the Microsoft Kinect for assessment of postural control. *Gait and Posture* 36, 3 (2012), 372–377. DOI:http://dx.doi.org/10.1016/j.gaitpost.2012.03.033

Alexandra Crabb. 2013. PrimeSense^{TM} Unveils Capri, World's Smallest 3D Sensing Device at CES 2013. (2013). http://www.primesense.com/news/primesense-unveils-capri/

Julian Dibbell. 2011. *Controlling computers with our bodies*. Journal article. Retrieved July 25, 2013 from http://www2.technologyreview.com/article/423687/gestural-interfaces/

T. Dutta. 2012. Evaluation of the Kinect sensor for 3-D kinematic measurement in the workplace. *Appl Ergon* 43, 4 (2012), 645–9. DOI:http://dx.doi.org/10.1016/j.apergo.2011.09.011 Dutta, Tilak eng Research Support, Non-U.S. Gov't England 2011/10/25 06:00 Appl Ergon. 2012 Jul;43(4):645-9. doi: 10.1016/j.apergo.2011.09.011. Epub 2011 Oct 20.

M. Gabel, R. Gilad-Bachrach, E. Renshaw, and A. Schuster. 2012. Full body gait analysis with Kinect. *Conf Proc IEEE Eng Med Biol Soc* 2012 (2012), 1964–7. DOI:http://dx.doi.org/10.1109/EMBC.2012.6346340 Gabel, Moshe Gilad-Bachrach, Ran Renshaw, Erin Schuster, Assaf eng Clinical Trial 2013/02/01 06:00 Conf Proc IEEE Eng Med Biol Soc. 2012;2012:1964-7. doi: 10.1109/EMBC.2012.6346340.

H. Gonzalez-Jorge, B. Riveiro, E. Vazquez-Fernandez, J. Martínez-Sánchez, and P. Arias. 2013. Metrological evaluation of Microsoft Kinect and Asus Xtion sensors. *Measurement* 46, 6 (2013), 1800–1806. DOI:http://dx.doi.org/10.1016/j.measurement.2013.01.011

Hal Hodson. 2013. Leap Motion hacks show potential of new gesture tech. *New Scientist* 218, 2911 (2013), 21. DOI:http://dx.doi.org/10.1016/S0262-4079(13)60864-7

Kourosh Khoshelham and Sander Oude Elberink. 2012. Accuracy and Resolution of Kinect Depth Data for Indoor Mapping Applications. *Sensors* 12, 2 (2012), 1437–1454. http://www.mdpi.com/1424-8220/12/2/1437

H. König. 2005. Access to persistent health information objects: Exchange of image and document data by the use of DICOM and HL7 standards. *International Congress Series* 1281, 0 (2005), 932–937. DOI:http://dx.doi.org/10.1016/j.ics.2005.03.186

Marketsandmarkets.com. 2013. Gesture Recognition & Touchless Sensing Market (2013 - 2018): By Technology (2D, 3D, Ultrasonic, IR, Capacitive); Product (Biometric, Sanitary Equipment); Application (Healthcare, Consumer Electronics, Automotive); Geography (Americas, EMEA, & APAC). *Market Analysis* (2013). http://www.marketsandmarkets.com/Market-Reports/touchless-sensing-gesturing-market-369.html

Microsoft. 2013. Kinect for Windows SDK. (July 2013). Retrieved August 17, 2013 from www.microsoft.com/kinect

Rachel Nuwer. 2013. Armband adds a twitch to gesture control. *New Scientist* 217, 2906 (2013), 21. DOI:http://dx.doi.org/10.1016/S0262-4079(13)60542-4

OpenNI. 2013. OpenNI framework. (2013). Retrieved July 25, 2013 from www.openni.org

PointCloud. 2013. PointCloud Library documentation. (2013). Retrieved July 2, 2013 from http://docs.pointclouds.org

Stephen Prentice and Adib Carl Ghubril. 2012. Hype Cycle for Human-Computer Interaction. *Gartner* (2012).

Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*. http://www.pointclouds.org/assets/pdf/pcl_icra2011.pdf

Todor Stoyanov, Rasoul Mojtahedzadeh, Henrik Andreasson, and Achim J. Lilienthal. 2012. Comparative evaluation of range sensor accuracy for indoor mobile robotics and automated logistics applications. *Robotics and Autonomous Systems* (2012). DOI:http://dx.doi.org/10.1016/j.robot.2012.08.011

Zeev Zalevsky, Alexander Shpunt, Aviad Maizels, and Javier Garcia. 2007. Method and System for object reconstruction, Patent nr. WO2007043036. (2007). Retrieved April 23, 2013 from http://patentscope.wipo.int/search/en/WO2007043036

Software Engineering Solutions for Improving the Regression Testing Methods in Scientific Applications Development

BOJANA KOTESKA, LJUPCO PEJOV AND ANASTAS MISHEV, University SS. Cyril and Methodius

The aim of this paper is to improve the testing process of scientific applications by proposing some software engineering solutions for regression testing. Given the fact that changes are a common part of the development of scientific applications the need for regression testing is essential. Concerning to improve the methods of regression testing the relationship between requirements and test case selection for regression testing is included as a relevant for improving the overall quality of the testing process. Special emphasis is given on the requirements' changes during the application development process and their impact on test case modifications. In addition, we also state the reasons for the implementation and the importance of the regression testing as a part of the verification process of scientific applications. In order to get more relevant results we made an interview with a scientist (chemist) about the regression testing practices in scientific applications. The conclusions and recommendations in this paper are based on the analysis of the results of a survey conducted among scientists in the HP-SEE project and the answers of the interview questions.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification —*Validation, Formal Methods*; G.4 [**Mathematics of Computing**]: Mathematical Software—*Certification and testing, Documentation, Verification*

General Terms: Verification

Additional Key Words and Phrases: Regression testing, scientific application, software engineering, software quality

1. INTRODUCTION

Advances in research in scientific areas continually impose the need for creating scientific applications. One of the definitions of scientific applications describes the scientific application as a mathematical model that simulate the natural phenomena [PcMag 2013]. The need for creating large and complex mathematical models mathematical calculations increases the probability of errors and thereby increases the need for performing changes in different parts of the application. Successful testing after a change ensures that the system is in a stable condition and it provides assurance that the system did not lose any functionality, but only new functionalities are added or the incorrect ones are replaced. Taking into account the fact that changes are a common part of a scientific application development process because of the constant changes in requirements and source code the need for regression test-

This work was supported in part by the European Commission under EU FP7 project HP-SEE (under contract number 261499).
Author's address: Bojana Koteska, University SS. Cyril and Methodius, Faculty of Computer Science and Engineering, Rugjer Boshkovikj 16, P.O. Box 393 1000, Skopje; email: bojana.koteska@finki.ukim.mk; Ljupco Pejkov, University SS. Cyril and Methodius, Faculty of Natural Science and Mathematics, P.O. Box 1001, 1000 Skopje; email: ljupcop@iunona.pmf.ukim.edu.mk; Anastas Mishev, University SS. Cyril and Methodius, Faculty of Computer Science and Engineering, Rugjer Boshkovikj 16, P.O. Box 393 1000, Skopje;email: anastas.mishev@finki.ukim.mk.

In: Z. Budimac (ed.): Proceedings of the 2nd Workshop of Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA), Novi Sad, Serbia, 15.-17.9.2013, published at http://ceur-ws.org

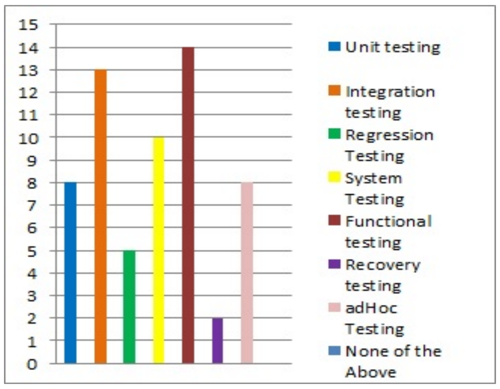


Fig. 1. Answers to the question: ”Which types of testing do you perform?”.

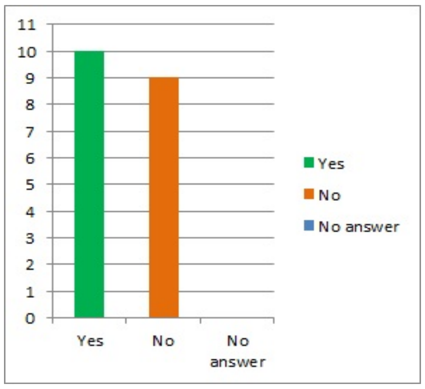


Fig. 2. Answers to the question: ”Do you link requirements to tests that verify them?”.

ing is essential [Letondal and Zdun 2003]. For example, the results obtained from physical experiments can be the reason for making changes in the requirements and application source code.

The main motivation for this research comes from the results obtained from the survey we conducted among 20 scientists in the HP-SEE [Koteska and Mishev 2013] (High-Performance Computing Infrastructure for South East Europe’s Research Communities) project. The scientific applications in this project are organized in three scientific research communities: Computational Physics, Computational Chemistry and Life Sciences. There are total 23 applications [HP-SEE 2013].

The purpose of this paper is to improve the testing process of scientific applications, especially the regression testing. Some of the most common software engineering practices are pointed out and several recommendations about each one are given. The main contribution is that we adapt some existing software engineering practices which are used for commercial software development where real customers exist by making small modifications. The difference between scientific and commercial software testing is in the testing methods. For example, one of the key differences is the definition of requirements and inability the both kind of software to be tested in the same way. Given that in the scientific software development, real customers do not exist, checking the accuracy of scientific applications is very difficult. Usually, to check whether the software satisfy a requirement, it is necessary to perform a real experiment [Segal].

The most important topics are the regression testing and the relationship between requirements and tests. In order to show the current practice of scientists in the project the next two questions were chosen as the most relevant to these issues. The results to the question about testing types which scientists performed in the application development process are shown on Fig.1. They show that most of the scientists performed integration and functional testing, but only five development teams performed regression testing and two teams perform recovery testing. Fig. 2 presents the results of the question about linking requirements and tests. The majority of the scientists did not define requirements or define them in inappropriate form therefore the fact that almost half of the development teams used to link requirements to tests is questionable.

2. BACKGROUND

Regression testing is defined as a repeated execution of test suites in order to ensure that the system does not fail after a modification and that test suites still pass [Christensen 2011]. In [Singh et al. 2010], the authors describe some regression testing techniques. They mentioned three categories of regression testing techniques: selection, test prioritization and hybrid techniques. Selection techniques choose some tests from the old test suite and execute them on the new modified version of the software application. Test prioritization reorders test suite in order to improve the effectiveness of testing. Hybrid techniques combine both the selection and test prioritization techniques. An approach for prioritization of test cases that should be used in regression testing process is presented in [Srikanth et al. 2005]. Test case prioritization is based on four factors: requirements volatility, customer priority, implementation complexity, and the fault proneness of the requirements. This approach is not fully applicable to scientific software development because real customers do not exist. The factor ”customer priority” must be removed. A test suite prioritization algorithm is given in [Srivastava et al. 2008]. This algorithm prioritizes the test cases with a goal of maximizing the faults that can occur during the source code execution. In [Remmel et al. 2012], the authors proposed a regression test environment, an infrastructure for testing, at system testing level for DUNE (the Distributed and Unified Numerics Environment). DUNE is a free software framework for solving partial differential equations with grid based methods [Bastian et al. 2008]. They create tests which support algorithm verification and scientific validation. In [Kelly and Sanders 2008], the authors mention the regression testing as a factor for improving the quality of scientific software development. An advice for turning the bugs into test cases is given in [Aruliah et al. 2012]. This approach can be used for building regression tests. The bad regression testing practices are presented in [Sanders 2008]. According to this paper, scientists usually do not perform systematic regression testing, but they use ad-hoc regression testing. Also, they use the same input data for regression tests for more than a year. The regression tests that are consisted of comparison of simulation results with reference results are not practical according to [Remmel et al. 2011]. They are used for ensuring the experimental results, but they should be avoided because the differences in the outputs may be caused by an enhancement in the code. According to the survey results presented in [Heaton et al. 2012]], regression testing is important for scientists, but they do not perform it correctly because they are not familiar with it. Also, the developers do not use automated regression testing. The other thing important for this research is the connection between the requirement changes and test case selection for regression testing. An approach to regression test selection which uses system requirements and their associated test cases instead of source code is presented in [Chittimalli and Harrold 2008].

3. THE INTERVIEW

In order to get more relevant results, we did an interview with a scientist (chemist) who has big experience in developing scientific applications and one of the participants in the HP-SEE project. There are total 12 questions that mostly refer to requirements changes, test cases and regression testing. The short version of the interview is presented in this section.

Q1: How often do changes in requirements occur during the application development?

A1: There are two possible reasons for changes in requirements in the course of application development. The first one is related to the new scientific issues that might have arisen during the research, the second one, on the other hand, comes into play when the researcher comes to (or invents) a new approach which is superior to the one that has been used up to that point.

D1: Advances in research and new results obtained from experiments are the most common reasons that lead to changes in scientific applications. Scientists know that the possibility of such changes is great. Consequently, it is necessary to make changes in several segments of the code, while not breaking parts that perform correct calculations. Also, it is necessary to keep the concept of the functioning of the system as a whole.

Q2: How do changes in requirements affect the source code?

A2: If only a minor change in the algorithm or addition of another variable that needs to be computed is required, often small changes in the source code are required. On the contrary, when a new algorithm or approach is intended to be included in research, often more profound changes to the source code are required, and it certain cases it is much easier to write the code from scratch than to introduce changes in the current version.

D2: Problems faced by scientists when they need to make major changes in the algorithm commonly occur because of an inadequate documentation and poor visibility of the source code. Sometimes they do not understand the code that had developed before a certain period of time. Therefore, it is easier for them to develop a new algorithm for solving the same problem.

Q3: When do most of the changes occur (at which stages of development)?

A3: After the code has been tested on a wide variety of systems. In this phase, often most of the "bugs" are identified, or some errors (both conceptual and trivial) are seen. However, also numerous changes can occur at any point at which a change in requirement occurs.

D3: Usually scientists perform testing after a certain period of time when there are many written lines of code. This approach causes errors to be discovered too late, and thus increases the number of changes that need to be made. Small changes which are made as a result of a change in any one of requirements that are not specified in the appropriate form, are not recorded usually.

Q4: How much time do you need to make a source code correction after each change made?

A4: It depends on the complexity of the correction, addition, or functionality enhancement. In most cases, especially if only minor changes are required, source code modifications are quick. In cases, however, where new conceptual changes are introduced, or new functionality in the code is included, the source code changing may take an appreciable amount of time.

D4: Changes that occur as a result of the addition of new functionalities in the code or change in the conceptual model are usually time-consuming. In particular, this happens if the code is infinite, badly written and undocumented.

Q5: Do you perform regression testing ? If yes, how do you perform it?

A5: I often do, unless the change is exceptionally small or pretty trivial. I usually repeat certain calculations on some sample systems or on cases which I have treated before, or some prototype systems (e.g. analytically-solvable systems).

D5: Scientists usually preform testing after changes, but not always on time. For example, several changes can be made in the source code and after that when testing is performed an error can be found. In this case, it can be difficult to determine the reason for its occurrence.

Q6: Do you avoid regression testing after the change and if you avoid it, why?

A6: I usually do not insist on avoiding regression testing (unless the change is really trivial, e.g. improved value of a fundamental or other constant).

D6: This testing method is usually used by scientists, but the changes are not documented and a previous version of the application is not kept. Sometimes testing after a big change can give accurate results, but later it may be necessary to return to the previous stable version of the application.

Q7: Do you make test case change after changing application test requirements associated with those test cases? If not, why?

A7: Changes in test cases are required if there are substantial changes in the functionality of the code, e.g. if the code applicability is extended to a more flexible group of systems, or if certain new concepts are introduced.

D7: Scientists usually do not create test cases for each requirement because they do not write a proper requirements specification. If a change in a requirement occurs and it is not propagated in the corresponding test cases, then the possible errors may not be noticed.

Q8: How do you make test cases?

A8: The test cases are usually either simple, prototypical systems, for which the required parameters are either known or can be computed by e.g. analytical approach.

D8: The answer shows that the tests are usually performed with the help of systems that already have known analytical solutions. This approach makes the process of specification of test cases difficult.

Q9: Does inadequate testing adversely affect the application development process?

A9: Sure, as this can affect both the code concepts and functionality.

D9: Scientists are aware that the wrong way of testing and insufficient testing can disrupt the functionality of the application. Improving the way of testing applications can greatly increase the quality of the applications.

Q10: Do you use any tools in the testing process and if you do, what type of tool it was?

A10: No.

D10: There are tools which create test cases, run tests, keep records of test cases and automate much of the work that is carried out manually. The lack of using testing tools during the development process can increase the time required for testing and time required for developing the application.

Q11: Do you record all the changes made to the requirements and the source code? Do you have any documentation about that? Do you use some templates?

A11: I only make comments throughout the source code. I usually make no documentation for my highly specialized codes. However, if one intends to make a more flexible code, for a more general purpose, then the actual usage of the code will be highly dependent on the quality of the documentation.

D11: Certainly, the main goal of the development process is not the creation of documents, but they serve as an additional support mechanism for dealing with changes in requirements and source code. There are a number of templates offered by software engineering which with minor modifications will fit perfectly in the development of scientific applications.

Q12: Do you think that regression testing is important?

A12: It certainly is. It can help in both the time line of the application development, and also in its correctness.

D12: The regression testing improves the quality of applications by ensuring that adding a new functionality or changing the existing will not impair the functionality and correctness of the application.

4. SOFTWARE ENGINEERING SOLUTIONS FOR REGRESSION TESTING

4.1 Definition of Requirements

Before the process of development starts, scientists need to define the application requirements. All requirements cannot be predicted in advance because it depends on the progress in research. Also, changes in requirements occur very often. A good practice is to define the requirements in iterations. The iterative development is recommended for the entire application development process because the requirements need not be completely understood and specified at the start of the development process and they can be added in any iteration [Jalote et al. 2004]. It is necessary to specify the expected results for each functionality. If it is not possible to predict the exact value, then it is necessary to define the range of values where the solution lies. A domain specific requirements model for requirements is presented in [Li et al. 2011]. The model includes the scientific knowledge as a key factor for identifying requirements and it provides domain specific requirement model elements and associations between them. should be specified using templates proposed by software engineering. Also, some modifications to the template should be made if necessary.

Each requirement should contain the following basic fields: Id - unique requirement identifier; Name - short name of the application that reflects the functionality that needs to be realized; Type of application - functional, non-functional or domain request; Version - current version of the requirement; Description - a detailed description of the functionality that needs to be realized (Description of the resources required to execute the application); History of the changes in each version of the requirement.

The process of defining requirements will help in the test cases specifications and in the process of testing the application. Well specified requirements lead to a more precise definition of the test cases and better organization of the overall development process.

4.2 Definition of Test Cases

Test cases can be generated automatically by using a static approach which generates inputs from some kind of model of the system (model-based testing) and a dynamic approach which generates test cases by executing the program repeatedly [Artho et al. 2005]. The main thing that needs to be considered while generating test cases is to create at least two test cases for each requirement. Moreover, the first described test case should return the correct value (the one that is expected), and the second described test case should return an incorrect value. Test cases where a correct value cannot be specified should contain a range of values.

Each of the descriptions of the test cases should include the following fields: Id - a unique test case identifier; Version - the current version of the test case; Name - name of the test case; Requirement id - id of the requirement that is connected to this test case; Goal -a description of the goals of the test case; Conditions - conditions that must be met in order to perform the test case (results from other tests or some additional conditions); Execution environment of the test case; Expected results; History of the changes in each version.

Research has shown that scientists do not use any additional tools that automate the process of testing and comparing the outputs with the expected results. This is particularly important when regression testing is performed because the same test cases are being repeated.

4.3 Test Cases Selection

The selection of test cases before the process of regression testing starts is one of the most important prerequisites for quality testing. The decision about test cases that should be selected and re-executed after a change depends on many factors. The selection of test cases that need to be performed again included those test cases that are related to the performed change. A good test cases selection technique eliminates the redundant test cases and assigns priority to test cases such that test cases with higher fault-detection capability are assigned a higher priority. Some of the most common selection techniques for procedural programs are: dataflow analysis-based techniques, slicing-based techniques, firewall-based techniques, differencing-based approaches, control flow analysis-based techniques [Biswas et al. 9 01].

The process of the execution of the relevant test cases can be split into three categories: test cases that relate to the requirement that had been changed; test cases associated with the module from the source code which had been changed; Execution of the test case that had been changed. The second option is the most difficult to be performed- to find the test cases when a change was made to the source code. Then the functionality of that part of the code needs to be identified and the appropriate test cases associated with it need to be found. For easier identification of test cases that relate to a particular change in the code, it is necessary to organize the code into smaller independent units - functions. A good selection technique of test cases requires proper specification of the application requirements and test cases, and maximum visibility and organization of the source code parts. Also, if a test case depends on other test cases and the change was performed in that test case it is necessary to execute all test cases that it depends on.

4.4 Regression Testing

Some of the main reasons for regression testing are: adding or changing a functionality and bug detection, fixing defects, adding or adapting the existing functionalities or porting the software to different environments [Biswas et al. 2010]. Another reason for regression testing is refactoring. In [Rachatasumrit and Kim 2012], authors showed the relationship between refactoring edits and regression testing by applying a change impact analysis and a refactoring reconstruction analysis.

A good approach is to use the hybrid technique defined in [Wong et al. 1997] which combines the modification, minimization and prioritization-based selection using a list of source code changes and the execution traces from test cases run on previous versions.

The process of regression testing requires several additional things that must be done before the process of testing starts. The activity diagram of the steps before and after the regression testing process when a new functionality should be added or an existing should be changed is shown on Fig. 3. After changing a requirement, when a new functionality is required or the existing one should be changed, first thing that needs to be done is test case correction. It means that all test cases affected by the requirement change must be modified. Also, the appropriate modifications should be done in the source code.

One of the most important thing is the selection of a set of test cases for regression testing. In this case, the changed test cases must be included in the set. After this step, the regression testing should be performed. If all test cases pass then the regression testing is performed successfully. If not, then all test cases changes should be reviewed and the next steps should be repeated.

The Fig. 4 presents the steps before and after regression testing when a bug in the application is found. After the bug is found, the reasons for that must be clearly indicated and adequate changes should be made in the source code. Before the testing is performed the selection of the test cases should be done. It is not good practice to take into consideration all test cases during the regression testing because not all test cases are affected by the found bug. If any of the tests fail then the reasons for the bug should be considered again and all steps after should be repeated.

The regression testing is a very important step of the development process of scientific applications. The realization of the regression testing mostly depends on the specification of requirements and test cases. Since when developing scientific applications is not possible to define all requirements at the beginning a good practice is the process of testing to take place in stages (iterations). Moreover in each iteration the requirements that have been implemented or changed should be tested. This type of testing helps in reducing the number of errors that occurred as a result of a change of requirement or a change in the source code of a scientific application. The main reason for the implementation of the regression testing is that it can improve the quality of the whole testing process and the application.

5. CONCLUSION AND FUTURE WORK

In this paper we have presented some useful software engineering solutions for regression testing. The results obtained from the survey we conducted among scientists in the HP-SEE project and the interview showed that the regression testing is a fundamental part of the scientific application development process. We gave an overview of the most common reasons for changes that occur during the development of the scientific applications. Also, the most important steps of the regression testing were presented. This paper outlines the software engineering practices that should be included in the development process in order to improve the quality of the testing process and overall scientific application.

Our future work is oriented to developing a framework for scientific application development, where more specific software engineering practices will be given for each development phase and our future

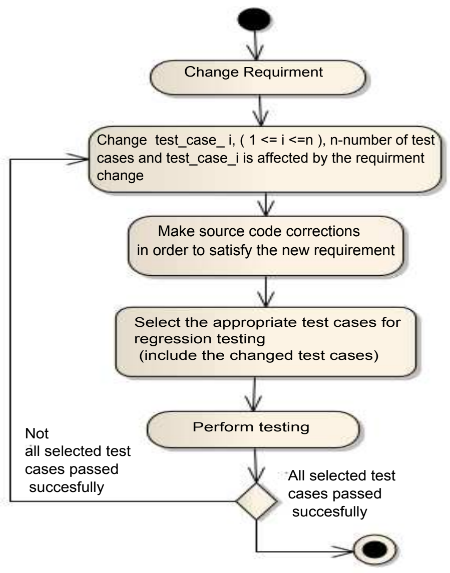

Change Requirment
Change test_case_ i, (1 <= i <=n), n-number of test cases and test_case_i is affected by the requirment change
Make source code corrections in order to satisfy the new requirement
Select the appropriate test cases for regression testing (include the changed test cases)
Perform testing
Not all selected test cases passed succesfully
All selected test cases passed succesfully

Fig. 3. An activity diagram of regression testing process when a new functionality is added or the existing is changed

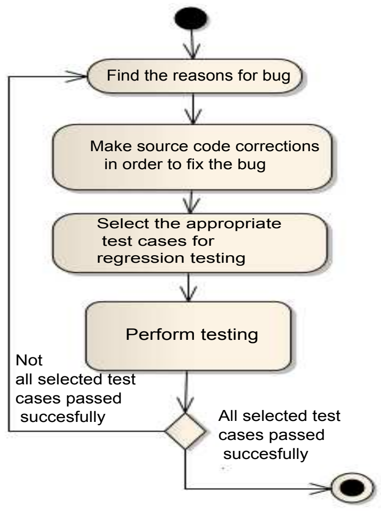

Find the reasons for bug
Make source code corrections in order to fix the bug
Select the appropriate test cases for regression testing
Perform testing
Not all selected test cases passed succesfully
All selected test cases passed succesfully

Fig. 4. An activity diagram of regression testing process when a bug is found

research is aimed at proving this concept in practice. It means that all these recommendations and practices will be used in the development process for a specific scientific application. The purpose of this framework is to provide a detailed guide for scientists when developing scientific software. It will give a detailed software engineering solutions for each development phase (defining requirements, programming, testing, etc.) and also some development techniques. It will help scientists to learn some common software engineering practices. The framework will include the software engineering practices for scientific software development, but also in order to adapt to the scientific software development process, some modifications of the existing software engineering practices for the commercial software development will be made.

REFERENCES

Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Mike Lowry, Corina Pasareanu, Grigore Rosu, Koushik Sen, Willem Visser, and Rich Washington. 2005. Combining test case generation and runtime verification. *Theor. Comput. Sci.* 336, 2-3 (May 2005), 209–234. DOI:http://dx.doi.org/10.1016/j.tcs.2004.11.007

D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Katy Huff, Ian Mitchell, Mark Plumbley, Ben Waugh, Ethan P. White, Greg Wilson, and Paul Wilson. 2012. Best Practices for Scientific Computing. *CoRR* abs/1210.0530 (2012).

P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klofkorn, M. Ohlberger, and O. Sander. 2008. A generic grid interface for parallel and adaptive scientific computing. Part I: abstract framework. *Computing* 82, 2 (July 2008), 103–119. DOI:http://dx.doi.org/10.1007/s00607-008-0003-x

Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. 2010. Regression Test Selection Techniques: A Survey. (2010).

Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. 2011-09-01. Regression test selection techniques: a survey.(Report). *Informatica* 35, 3 (2011-09-01), 289(33).

Pavan Kumar Chittimalli and Mary Jean Harrold. 2008. Regression test selection on system requirements. In *Proceedings of the 1st India software engineering conference (ISEC '08)*. ACM, New York, NY, USA, 87–96. DOI:http://dx.doi.org/10.1145/1342211.1342229

H.B. Christensen. 2011. *Flexible, Reliable Software: Using Patterns and Agile Development*. Taylor & Francis. http://books.google.mk/books?id=VQaf_vOTDzMC

Dustin Heaton, Jeffrey C. Carver, Roscoe Bartlett, Kim Oakes, and Lorin Hochstein. 2012. *The Relationship between Development Problems and Use of Software Engineering Practices in Computational Science & Engineering: A Survey*. Technical Report SERG-2012-05. Department of Computer Science, The University of Alabama. http://software.eng.ua.edu/reports/SERG-2012-05

HP-SEE. 2013. HP-SEE officail website. http://www.hp-see.eu/. (2013). Accessed March 22, 2013.

Pankaj Jalote, Aveejeet Palit, and Priya Kurien. 2004. Timeboxing: A process model for iterative software development. *Journal of Systems and Software* 2004 (2004), 3.

Diane Kelly and Rebecca Sanders. 2008. Assessing the Quality of Scientific Software.

Bojana Koteska and Anastas Mishev. 2013. Software Engineering Practices and Principles to Increase Quality of Scientific Applications. In *ICT Innovations 2012 (Advances in Intelligent Systems and Computing)*, Smile Markovski and Marjan Gusev (Eds.), Vol. 207. Springer Berlin Heidelberg, 245–254. DOI:http://dx.doi.org/10.1007/978-3-642-37169-1_24

Catherine Letondal and Uwe Zdun. 2003. Anticipating Scientific Software Evolution as a Combined Technological and Design Approach. In *in Second International Workshop on Unanticipated Software Evolution (USE2003*. 1–15.

Yang Li, N. Narayan, J. Helming, and M. Koegel. 2011. A domain specific requirements model for scientific computing: NIER track. In *Software Engineering (ICSE), 2011 33rd International Conference on*. 848–851. DOI:http://dx.doi.org/10.1145/1985793.1985922

PcMag. 2013. Definition of scientific application. http://www.pcmag.com/encyclopedia/term/50872/scientific-application. (2013). Accessed March 20, 2013.

N. Rachatasumrit and Miryung Kim. 2012. An empirical investigation into the impact of refactoring on regression testing. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. 357–366. DOI:http://dx.doi.org/10.1109/ICSM.2012.6405293

Hanna Remmel, Barbara Paech, Peter Bastian, and Christian Engwer. 2012. System Testing a Scientific Framework Using a Regression-Test Environment. *Computing in Science and Engg.* 14, 2 (March 2012), 38–45. DOI:http://dx.doi.org/10.1109/MCSE.2011.115

Hanna Remmel, Barbara Paech, Christian Engwer, and Peter Bastian. 2011. Supporting the testing of scientific frameworks with software product line engineering: a proposed approach. In *Proceedings of the 4th International Workshop on Software Engineering for Computational Science and Engineering (SECSE '11)*. ACM, New York, NY, USA, 10–18. DOI:http://dx.doi.org/10.1145/1985782.1985785

R. Sanders. 2008. *The Development and Use of Scientific Software*. Queen's University (Canada). http://books.google.mk/books?id=9L7jKBqZpm0C

Judith Segal. Scientists and software engineers: A tale of two cultures. http://oro.open.ac.uk/17671/

Yogesh Singh, Arvinder Kaur, and Bharti Suri. 2010. A Hybrid Approach for Regression Testing in Interprocedural Program. *JIPS* 6, 1 (2010), 21–32.

H. Srikanth, L. Williams, and J. Osborne. 2005. System test case prioritization of new and regression test cases. In *Empirical Software Engineering, 2005. 2005 International Symposium on*. 64–73. DOI:http://dx.doi.org/10.1109/ISESE.2005.1541815

Praveen Ranjan Srivastava, Krishan Kumar, and G. Raghurama. 2008. Test case prioritization based on requirements and risk factors. *ACM SIGSOFT Software Engineering Notes* 33, 4 (2008).

W.E. Wong, J.R. Horgan, S. London, and H. Agrawal. 1997. A study of effective regression testing in practice. In *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*. 264–274. DOI:http://dx.doi.org/10.1109/ISSRE.1997.630875

The Mobile Product Line

LUKA PAVLIČ, University of Maribor

In this paper we present the challenges during the simultaneous development of multiple mobile application versions with different sets of functionalities. Some of them are core, the other optional, and the third alternative. One of the indicated appropriate solutions is the approach of software product lines. In this paper we presents practical experiences during the product line implementation in the case of mobile applications for the Android operating system. It has at least six to eight different versions simultaneously available. Among others, these are freely available version, paid version, development, test and demonstration version. We also offer certain versions for the BlackBerry phones, some mobile application components share common functionalities with a portal server.

General Terms: Software Engineering

Additional Key Words and Phrases: Software product lines, Mobile applications, Mobile product lines, Android

1. INTRODUCTION

Software development has always been pursuing some universal business goals in the context of the business environment. These include high-quality software, rapid development, fast market penetration, low development costs, low maintenance cost, the possibility of rapid changes and so on. Often we are faced with software development projects, where the product is just one piece of specialized software. Such project would definitely not pursue above mentioned business goals sufficiently. In the scope of software reuse, software engineering has already given some answers on how to enable rapid development, low cost and high quality at the same time. One of these answers can be found in the software product lines (SPL). In this paper we will present practical experience with the introduction and successful implementation of the SPL approach in mobile solution @life (www.a-life.eu.com). In the second section we present the theoretical foundations of the SPL in terms of motivation, attitudes and acquisitions. The third section gives some practical aspect of the SPL for mobile applications for the Android operating system.

2. SOFTWARE PRODUCT LINES

2.1 Motivation: Gains from the Reuse Point of View

Reuse as one of the fundamental disciplines of software engineering plays an important role in the development of new, or maintenance of existing systems. It is almost impossible to have a software, that would result in a single version. One of the typical example of a well-established and diversified versions by the same piece of software are widely available commercial packages that offer a different set of functionalities according to the users' needs and/or financial investment. E.g. publishing software packages are available to private users, entrepreneurs or large organizations. Software could also be tailored to a specific set of hardware. After all, we are talking about diversity even when we have at the declarative level only one version of software. Even in this case we still need to have test environment software, production environment software etc.

These challenges are addressed with reuse approach. The reuse in software engineering has always dictated development models. From modular development in the sixties and seventies, through the object, component and later service-oriented development.

While developing a set of similar software products, separated by only a certain small set of specialized functionality for individual products, with the possibility of different implementations of the same

Author's address: L. Pavlič, Institute of informatics, Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova ulica 17, 2000 Maribor, Slovenia; email: luka.pavlic@uni-mb.si.

In: Z. Budimac (ed.): Proceedings of the 2nd Workshop of Software Quality Analysis, Monitoring, Improvement and Applications (SQAMIA), Novi Sad, Serbia, 15.-17.9.2013, published at http://ceur-ws.org

functionality, we can do this without the approaches of reuse, or just with the appropriate use of established reuse methods: component development, patterns, libraries etc. SPL in addition to the existing mechanisms of reuse allow some other levels of reuse - reuse at the level of larger software pieces. Besides reusing technical building blocks, these include also reuse of the procedures and rules, associated with the software. They include single analytics, planning and management of software development. This is how SPL enables lowering development costs and raise of quality at the same time. SPL approach could be implemented when some of the following issues occur as a result of the complexity of the software [Northrop, 2013]:

- we develop the same functionality for a variety of products and/or customers,
- the same change should be made in a number of different software products,
- he same functionality should behave differently depending on the final product,
- certain functionality can no longer be maintained, and so the customer has to move to newer version of the software,
- we can not estimate the cost of transferring certain features to another software,
- certain basic infrastructure changes lead to unpredictable behavior of dependent products,
- the majority of effort is put into maintenance, and not the development of new functionality.

At this point, we can draw a parallel with other engineering disciplines, where such approach (product lines) are not only present, but also fully implemented. E.g. the automotive industry: based on the same chassis manufacturers produce a range of different vehicles within a common family. Certain vehicles may have a transmission made in the version of the automatic gearbox, certain vehicles can have the accelerator pad and the brakes also on the passenger side (for driving schools), the third type of the same family of vehicles offers a rather large backpack, third row of seats etc. The chassis of all these vehicles is not developed with the idea of completely universal, reusable element, but as a basis for known variability of vehicles. Thus, for example, the chassis do not offer the possibility of two pairs of rear wheels, as it is already clear from the beginning that the chassis was developed for four-wheeled vehicle. In addition to cost-effectiveness, such approach helps to achieve higher and, more importantly, more manageable quality. You can also have a separate test line, where you can develop improvements independently with possibility of introduction in production later. When, for example, improving rear door locking mechanism, the improvement may automatically reflected on all vehicles with rear doors, regardless of whether it is a commercial vehicle for the driving school, etc.. without any influence to the process of manufacturing or even undermine the basic plan family cars.

An example of this can be directly brought on the software using a software product lines. Software product lines mean just that: a common basis that share entire family of products. In doing so, we want to have also common support and development activities, such as planning, quality control, production engineering packages, unified debugging, etc..

One of the definitions of software product lines is as follows [SEI, 2013]:

A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

Software product lines provide on the one hand lowering the development costs, and on the other hand higher quality, both because of well-known common points of product family. Such an approach requires the development of appropriate architecture with possibility of introducing variabilities per product.

Nevertheless, the product line also require additional costs: architecture, building blocks and individual tests should include the possibility of variability, business plans must be made for multiple products, not just one, etc. However, in the long term the contributions of software product lines proved to be as follows [Northrop, 2013]:

- up to 10x improved productivity,
- up to 10x improved quality,
- joint development costs reduced up to 60%,
- shortened time-to-market up to 98%,
- the possibility of moving to new markets is measured in months, not in years.

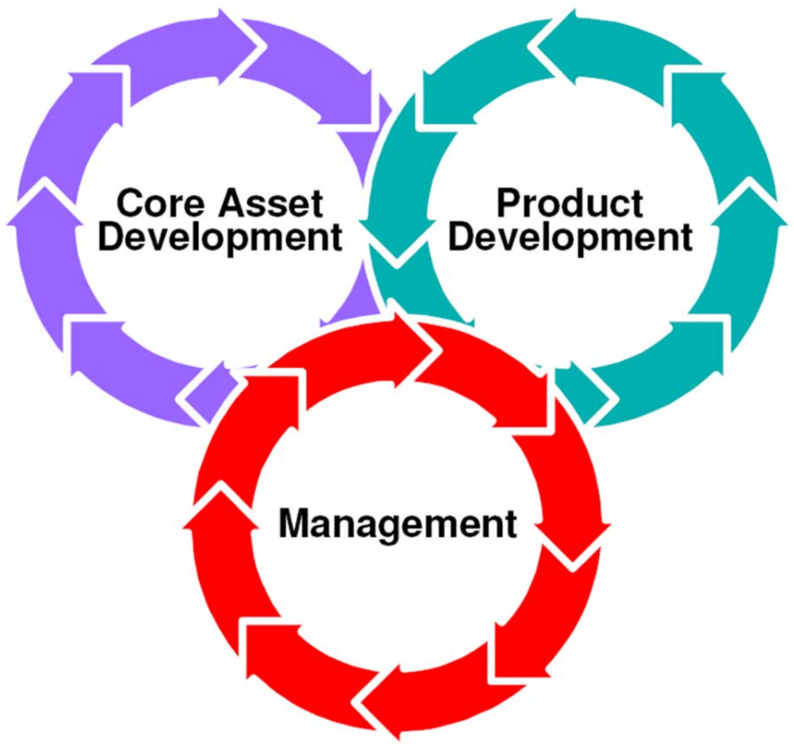


Fig. 1. Development activities, according SPL [Northrop, 2002]

As shown in Figure 1, the SPL approach separates the development of basic building blocks and their assembly into finished products [Northrop, 2002]. Within the development iterations we have to combine separate building blocks into product. The development of the basic product is thus completely separated from the assembly, while the assembly into the final product is completely automated. Management activities include supporting technical and organizational tasks.

It is also important to clarify, for which examples SPLs are not suitable. As a product line approach cannot be understood [Muthig, 2004]:

- reuse in general, typical of small general purpose building blocks,
- development of a unified system through reuse (libraries, components, services, samples, etc..),
- classic component or service-oriented development,
- architecture with the possibility of configurable behavior,
- versions of certain software.

2.2 Variability in Software Product Lines

Different authors give different definitions of variability. One of the most easy to read goes as follows:

Variability is the ability of software to be effectively extended, modified or adapted to use in a particular context. [Cavalcanti, 2013]

Keeping variation (difference or equality) in the individual building blocks of the final products is the basis of software product lines. In addition to the presence or absence of a functionality in a particular line we also often have specific functionality realized differently according to the line. These differences are recorded in the modeling process variability, which is an integral part of the system analysis. During the analysis we identify variable points, which later results in different realization of building block and final software:

- Functionality presence: If the functionality is present in all the lines and in all with the same realization, such functionality may be realized in the most general common building block.
- The lack of functionality: the functionality is not present in particular lines. In the case that the functionality is required in only one line, the functionality may be realized which in the line itself, otherwise it is necessary to introduce specific building block.

- A different realization: the functionality is available, but the realization will be different in different product lines. Different realization can be realized in line, unless the same feature can be found in multiple lines - in this case, it is reasonable to introduce a new building block, which is a specialization of the existing one.

Variability should be taken into account at all levels of analysis, design and development. In addition, it is necessary to pay special attention to the variability even during development in the context of management activities. Identification of variability is one of the main challenges of product lines. Other challenges are related to the realization of variability, their inclusion in the product line and adequately control variability. In addition, the possibility of variation functionalities (identified the variability in the level of functionality) on the technical level are mapped to a number of possible types of variability: [Muthig, 2004]

- variability of the data,
- variability in the level of implementation applications,
- variability at the level of the technology (software or hardware),
- variability in the level of quality criteria,
- variability of the target environment.

The variability technical realization is based on already established and well-known concepts in software engineering. To name a few [Clements, 2005]:

- building block inclusion, e.g. components, services,
- use of design patterns (factory, abstract factory, bridge, etc),
- changing behavior through inheritance,
- changing behavior with plugins,
- parameterization,
- configuration with deployment descriptors,
- directives at compile time,
- generating source code.

The best technical solution for realizing of variability will certainly contain proven and standard procedures in software engineering. It will also provide long-term stability in means of development and maintenance.

3. PRACTICAL EXPERIENCES – SOFTWARE PRODUCT LINE FOR ANDROID APPLICATION

3.1 @life Mobile Software Product Line

@life (www.a-life.eu.com) is a solution for people who need targeted support in detecting, monitoring and eliminating the negative effects of stress. It is intended for individuals in strengthening and upgrading health reserves and as such focuses on a healthy lifestyle.

Within the @life ecosystem, we have web, mobile, tablet and desktop applications. They are integrated with the @life cloud.

The mobile application has relatively large number of functionalities. They include:

- stress-assessment questionnaire,
- guiding physical tests,
- manage measurements, such as measurements of weight, blood sugar, blood pressure, etc..,
- perform advanced measurements of heart rate,
- guided exercises for strength, flexibility, balance, supported by videos and instructions,
- guided psychological exercises, such as relaxation, autogenic training, breathing exercises, etc..,
- performing outdoor activities and record various parameters such as heart rate, distance, altitude, calories consumption, speed, etc..,
- guided activities, such as interval training,
- activities diary, supported by analyzes in the form of graphs and geolocation services,
- measurements diary in the context of advice, based on deviations according to the desired value,
- guided psychological, physical or medical programs,
- etc.

Different set of features and a different behavior of the same functionality for different users was key to development method selection: software product lines. Thus, current application at the time of selection has become the basic building block upon which the rest of the line. Current product lines are visible in Figure 2

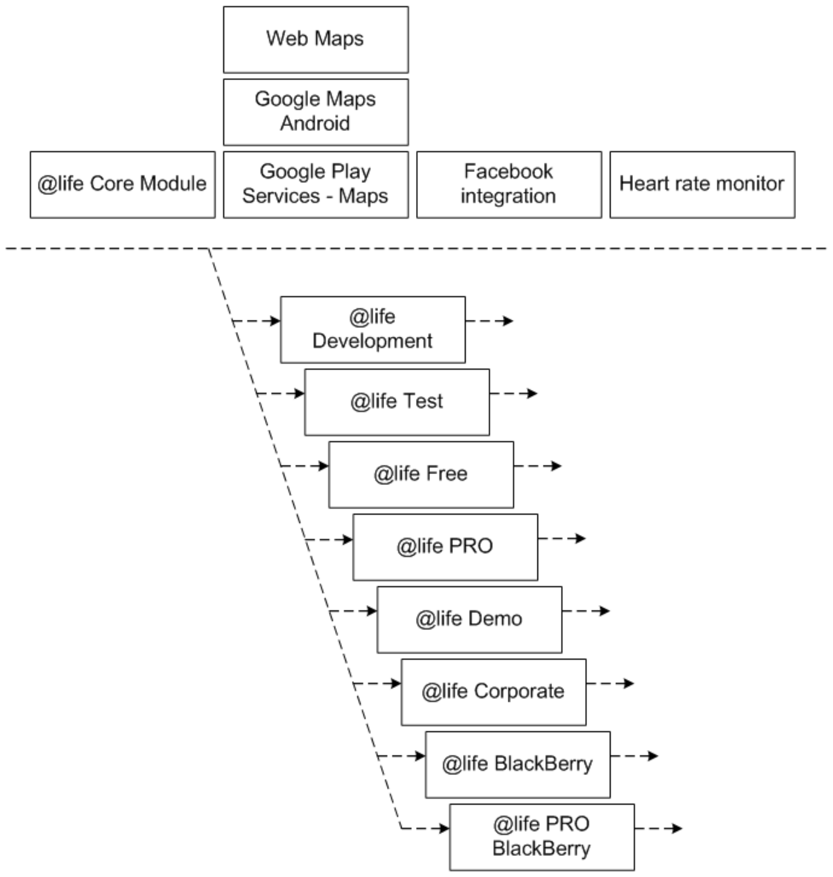


Fig. 2. @life product lines

The basic @life building block is fully running Android application. It contains only standard-based components (Android API). It realizes the functionalities that are common to all lines. At the same time, this building block also contains common architecture of mobile applications and the variability points of additional functionalities. All the basic building blocks and lines are fully operational applications - in short, it is not a library or component in the classic sense of the word. Applications within the product line are built into the final products with standard tools and Ant scripts.

Let us discuss some examples of optional and alternative features from both conceptual as well as technical point of view. One of the most illustrative examples of alternative functionality is the map that the BlackBerry phones is not supported in the form of a Google map.
Examples of optional features are:

- programs (not available in the free version),
- reminders (not available in the free version),
- exercises, videos (not available in the free version),
- demonstration and test versions expire after one month of production,
- etc.

Here are also many alternative functionalities. Let us mention only a few of the most illustrative examples:

- during the synchronization with the @life cloud, free versions do not include psychological and kinesiology exercises, nor anti-stress programs,

- test production line is connected to the test cloud, while the rest (except Corporate) is connected to the production cloud,
- current test version shows the maps using 3D acceleration eith the possibility of rotating the map, etc.., The production version contains the first generation of Google Maps, BlackBerry line offer only online map views,
- pro versions allow additional settings, e. g. fine tuning the automatic pause during activity.

There are variabilities also from language point of view. All applications are supporting six different languages. Currently we have also two more (demo) product line - one with the Chinese language support and one in Russian.

3.2 Practical examples of dealing with variabilities

The introduction of software product lines in the development of mobile solutions for @life was a challenge both from an organizational as well as technical point of view. During requirements gathering, designing and testing, functionalities were collected in a multi-dimensional table. That it was right decision also from technical aspect was proved especially because of several example situations:

- debugging and repairing found bugs,
- introduction of new product lines,
- inserting multilingual translations for several product lines.

When a quality assurance team found certain inconsistency or conflict with the specifications we had to fix it. In the case that there is an error in shared functionality, fixing error once automatically meant fixing error in all connected lines. Because of correct product line implementation we also did not expect any side effects while fixing errors, e.g. introducing new errors while fixing existing ones. The product lines has actually shorten the time of development, maintenance and troubleshooting.

Properly established architecture and software product line also allows virtually instant creation of new lines. We have to put a lot of effort to establish the appropriate architecture and variability points, which was SPL disadvantage. But in our case, this effort has repeatedly returned.

In terms of technical solutions for variabilities, we used the well-known best practices and approaches in software engineering area. These include the use of inheritance, extensions, and the component parameterization. Design patterns are also used heavily: e.g. factory, abstract factory, factory method, bridge, bean, etc..

We have used all three possibilities of variabilities in our product lines:

- include special features,
- remove unwanted features,
- changing behavior.

The inclusion of specialized functionality in the individual lines was achieved in several ways:

- preparation of the expansion point in the basic application (abstract methods, which expect concrete implementations in product lines - e.g. geolocation services),
- inheritance of existing classes and adding new methods calls in product line (e.g. calendar with reminders),
- using abstract factory pattern, which combines the functionality of the new line and its own user interface.

Exclusion of unwanted features was achieved mainly through inheritance and exclusion of unwanted features (such as not downloading programs for free lines), as well with the parameterization of the basic building blocks.

Changing behavior (e.g. line demo expires one month after construction), were achieved by appropriate design patterns, such as a bridge, factory method, builder, etc..

4. CONCLUSION

In this paper we reviewed the opportunities offered by the approach of software product lines. They were also presented in a case study on a specific project, the mobile solutions of @life. According to our own experiences, we can say that the described approach was correct and its positive effects are shown on a daily basis. We can confirm that the correct introduction of software product lines not only reduce development costs, but also significantly raise the quality. However, prior to the introduction of such approach it is necessary to keep in mind that SPL not only mean technical changes but also and mainly organizational changes and changes throughout the whole software development cycle.

REFERENCES

Software Engineering Institute, Software Product Lines, http://www.sei.cmu.edu/productlines, last visit may 2013.

Northrop, L. M. (2002). Sei's software product line tenets, IEEE Software 19(4): 32–40.

Northrop, L. M. Software Product Lines Essentials. Software Engineering Institute, http://www.sei.cmu.edu/productlines, nazadnje obiskano maj 2013.

Cavalcanti, Y. C., Machado, I. C., Anselmo, P. Handling Variability and Traceability over SPL Disciplines, Software product line – Advanced topic, Edited by Abdelrahman Osman Elfaki, Rijeka, 2013.

Clements, P. C., Bachmann, F., Variability in Software Product Lines, Product Line Practice Initiative, 2005.

Dirk Muthig, D. et. al., GoPhone - A Software Product Line in the Mobile Phone Domain, Fraunhofer Institut, Experimentelles Software Engineering, 2004.

Transforming Low-level Languages Using FermaT and WSL

DONI PRACNER AND ZORAN BUDIMAC, University of Novi Sad

There are many known problems in software maintanance, especially in cases where the available code is for whatever reason given only in low level executable versions. This paper presents a possible approach in understanding and improving such programs by translating it to an existing language *WSL* that enables the user to do formal, mathematically proven transformations of the resulting code. Such transformations can be done manually, but great improvements in the structure of the program can also be achieved by automatic scripts. Two prototype tools are presented that translate a subset of x86 assembly and a subset of Java bytecode, illustrated by examples showing the transformation process.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**] Distribution, Maintenance, and Enhancement

General Terms: Theory, Experimentation

Additional Key Words and Phrases: software evolution, FermaT, WSL, assembly, bytecode, transformations, translation

1. INTRODUCTION

One of the serious problems of modern software engineering is the perceived ageing of software. Although an application correctly written 20 years ago should still work as designed, maybe the underlying system is no longer available, making the application useless, or on the other hand maybe the user now needs a different result due to changes in the "real" world.

The problems of integrating legacy libraries, often available just as assembly code, into modern software/hardware systems can be tackled in different manners. One of the most efficient approaches, especially in the short term, is to encapsulate the functionality of reliable software[Sneed 2000]. Sometimes this is not really applicable – like in situations where new features need to be added to the system or, even worse, when there are bugs in the original software, in which cases it is necessary to understand and improve the original code.

The focus of this paper is on presenting two tools for working with low level code, that could help with understanding the logic behind the code and also potentially enable automatic restructuring of the code. One tool uses a subset of x86 assembly as its input, while the other one works with MicroJava bytecode. Both of the tools translate the programs into the high level language *WSL* that enables formally proven transformations on the source code, resulting in semantically equivalent code that should be much easier to understand.

The rest of the paper is organised as follows. Section 2 presents the existing transformation system that was used in this work. Section 3 shows the main steps and principles of the assembly translation and transformation process, as well as the tools created during this research. Following is an example

This work is partially supported by Ministry of Education and Science of the Republic of Serbia, through project no. OI174023: "Intelligent techniques and their integration into wide-spectrum decision support";
Author's address: Doni Pracner, Zoran Budimac, Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia; email: doni.pracner@dmi.rs zjb@dmi.rs

In: Z. Budimac (ed.): Proceedings of the 2nd Workshop of Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA), Novi Sad, Serbia, 15.-17.9.2013, published at http://ceur-ws.org

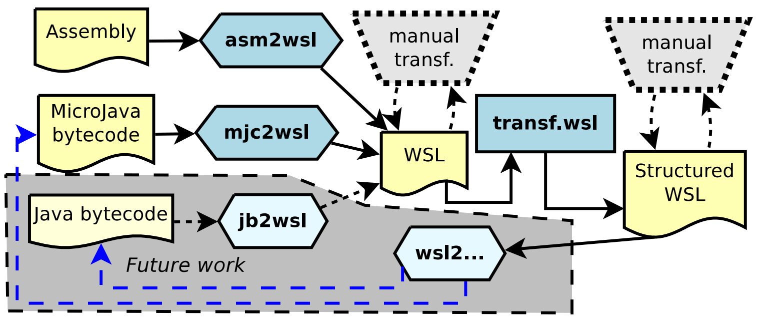


Fig. 1. Work-flow diagram of the tools – current and future

```
ACTION start:
start == code block 1 END
name1 == code block 2 END
name2 == code block 3 END
...
...
ENDACTIONS
```

Fig. 2. An action system

of the process, and some issues with the development. Section 4 introduces the bytecode tool, as well as illustrations of its work. Finally, conclusions, comparisons to related work and tools, as well as options for future work are given in Section 5.

2. WSL AND FERMAT

WSL (Wide Spectrum Language) is being developed by Martin Ward since 1989[Ward 1989]. A part of it is *MetaWSL* which gives the users constructs to write programs that will be able to transform code (internally represented as abstract syntax trees) using formal transformations. The current implementation is the *FermaT program transformation system*[Ward and Zedan 2005], and it is almost completely written in *MetaWSL*.

The main characteristics of the language is a strong mathematical core and the use of formal transformations, giving a reliable and provable system of improving software. The *wide spectrum* in the name means that there are constructs in the language that can be used for a wide spectrum of applications in development: from abstract specifications to low level program code.

The system was successfully used in many projects migrating legacy assembly code to maintainable C/COBOL code[Ward 1999; 2000; Ward et al. 2004]. There was also work on expanding the language to include support for concurrent programming[Younger et al. 1997] and object oriented programming[Chen et al. 2006], as well as incorporating a type system which improves the current state of transformations and provides a base for many future expansions[Ladkau 2009].

Action systems is a special structure in WSL which was specifically created to cope with unstructured jumps, which are very common in assembly code. It consists of a number of *actions* which can call each other, as shown in Figure 2. Once an action finishes it returns the control to the caller. Therefore an *action system* finishes when the start action finishes, or when a special, reserved action name “Z” is called which results in a momentary stop of the system.

Examples of action systems in use can be seen later in Figures 5 and 7.

3. ASSEMBLY TRANSFORMATION

Our transformation process consists of two basic steps: first we use our tool *Asm2wsl*[1][Pracner and Budimac 2011] to translate the assembly code to *WSL*, trying to capture all of the aspects of the original code without much effort to optimise at this step. Second, we use *Trans.wsl* (a script written in *WSL*) for automated transformations on the translated code. There is always a possibility to apply manual transformations, either before or after the automated transformations.

At its base, this approach is similar to those in the past that used WSL. But the difference is that our main goal in the process is to get a high level version of the original program that will represent all the aspects of its functioning. The approach using WSL presented in a number of papers [Ward 1999; 2004; Ward et al. 2004] creates additional files during the translation to WSL, which contain data about the variables and their mapping in the memory. These files are then used when the transformed and improved code is translated into (for example) *C* code and the appropriate pointer types are created.

Our approach should give a better understanding of the original code since we are looking at everything as high level structures, and it also enables us to run the translated programs directly in the WSL interpreter, without a need for an additional translator from WSL into another (semi) low level language. The downside is that assembler structures are often obscure, access data in different manners, and are therefore hard to understand and represent as high level, which limits the current version of the tools to a smaller subset of assembly code.

Asm2wsl is a tool that translates a subset of x86 assembly to *WSL*. Of course, being that there are many different flavours of assembly, and that it is very hard to automatically distinguish and adequately process them, a decision was made to focus on programs for a single type of assembler. The choice was the format first introduced in *Microsoft Macro Assembler (MASM)* that was consequently accepted by Borland's *Turbo Assembler (TASM)*[Borland International 1990], due to the familiarity with the later tool at our institution. To keep things simpler, at the moment it mostly presumes that we are working with an 80286 processor, the reason being that as they were developed, newer processors were mostly extended with more registers, options to work with bigger words and more specialised commands, which are not of great importance to the concepts we are translating.

The tool has been implemented in Java, making it platform independent and a good match to *FermaT*, which can also be run on a number of platforms. At its core, this is a line by line translator, with the focus on translating all aspects of the original code, without considering optimisation at this stage of the process. This generally results in programs that are much larger than the original assembly, but later on automatic transformations are able to reduce the size of the code. The same principle was successfully used in earlier translators which use WSL for transformations [Ward 2000; 2004; Ward et al. 2004].

Assembly commands work (more or less) directly with the processor. Being that high level languages do not do this, to capture all the aspects of these commands, we created a "virtual" processor. In it we have local variables to represent processor registers. Bits from the flag register are all defined as separate variables, which they practically are in the processor.

The processor can of course work with variables of different sizes, which introduces the need to work with them differently (the different scopes of values) and another problem – how to detect them? To handle this an additional overflow variable was introduced, and the translator tries to detect the size of the target variable from the original context. Based on the value the flag variables (in most cases overflow) are set like they would be in the real processor. An 80286 processor works with just two sizes: 8 and 16 bits, which is one of the reasons for presuming an older architecture in the study.

[1]the tool is available from the project's page http://perun.pmf.uns.ac.rs/pracner/transformations

The principle can then be tested on simpler examples and extended in the future as needed to bigger variable sizes.

In an *x86* processor *Low* and *High* parts of registers can be accessed independently (i.e. in 16 bits the lower and higher 8 bits, and analogue in bigger ones). Being that our goal were high level structures, we wanted to exclude direct memory operations, so these were implemented with additional operations that set the adequate parts of the register, at the same time preserving the potential side effects of the original code.

Labels in the original code are translated as *Action system* names (see Section 2). The whole system that we generate when we translate a "normal" assembly program is by nature *regular* (meaning that none of the calls ever return, that is, all of the actions just call other actions, and the system is finished with a CALL Z). Because of the special properties, these can be transformed easily into structured code.

Basic operations with arrays are also supported by the tool, with an automatic adjustment to the indexes, which is necessary since arrays start from 1 in *WSL*, and from 0 in assembly.

The processor's internal stack is implemented as a global list/array. The pop and push commands take and put elements on the start of this list directly. No additional checks (such as element size and compatibility, presence of elements on the stack) are performed, being that we presume to work with programs that worked correctly in their original form.

Macro structures are not translated at all at this stage of the development. On the other hand there are some special macro names that are recognised and translated directly into *WSL* code to enable input/output operations. For instance print_num x and print_str x are directly translated to PRINT(x). Similarly read_num and read_str are directly translated to *WSL* commands for reading numbers and strings, respectively.

The tool also has support for translating procedures from assembly. They are translated as nested Action Systems, so that local labels can be created, and it also enables us to return to the point of the original call once the procedure has finished its work. The translated programs worked from the interpreter without modifications. Transformations were also successful, despite the process resulting in action systems that are not *regular*. In the initial small tests, the procedures were simplified and then included in the main action system, as will be seen in an example in Section 3.

The second part of the process consists of a small program written is WSL that goes through the abstract syntax tree of the translated program, and tries to apply some of the available transformations on adequate nodes. All of the transformations implemented in WSL need to have procedures that will test if they can be applied to the given node, so this part of the code is relatively simple. For example, a transformation that unrolls a WHILE loop will (among other things) first check if the given node is in fact a WHILE statement.

Some of the important transformations are collapsing of the action systems into endless loops with exits in the middle and the subsequent transform of those into WHILE loops. At the same time constants are propagated through the code and various redundant parts are removed.

Example. This example is an illustration of how the transformation of procedures should work. *SumN* is a simple program with a call to a procedure that sums the top of the stack. The original assembly procedure is shown in Figure 3.

The whole assembly program is, more or less, just loading the data onto the stack and calling the procedure. As explained before (end of Section 3), the procedure will be translated into a nested action system, as shown in Figure 5.

Transformations are then applied to the obtained system – removing flags, collapsing action systems, and transforming the loops to WHILEs. The end result is the procedure transformed into a form shown in Figure 3, which is directly included into the main program inplace of its call.

```
sumn    proc
;take n from the top of the stack
;sum the next n top elements of the stack
        pop cx
        mov bx, 1
        mov ax, 0
        mov dx, 0
theloop:
        pop ax          ; get next from stack
        add dx, ax      ; array sum is in dx
        cmp bx,cx       ; is it the final one?
        je endp         ; skip to end if ti is
        inc bx
        jmp theloop
endp:
        push dx         ;result
        ret
sumn    endp
```

Fig. 3. Assembly version of the SumN procedure

```
cx := HEAD(stack);
stack := TAIL(stack);
ax := HEAD(stack);
stack := TAIL(stack);
WHILE bx <> cx DO
  dx := ax + dx;
  IF dx >= 65536 THEN dx := dx MOD 65536 FI;
  bx := bx + 1;
  ax := HEAD(stack);
  stack := TAIL(stack) OD;
stack := <dx> ++ stack
```

Fig. 4. SumN – transformed

```
ACTIONS A_S_start:
A_S_start ==
 ...... stack init etc ......
  stack := < n > ++ stack;
  CALL sumn;
  rez := HEAD(stack);
  stack := TAIL(stack);
  PRINT(rez);
  CALL end1
END
end1 ==
   CALL Z END
sumn ==
  ACTIONS dummysys:
  dummysys ==
    cx := HEAD(stack);
    stack := TAIL(stack);
    bx := 0;
    ax := 0;
    dx := 0;
    CALL theloop
  END
  theloop ==
    ax := HEAD(stack);
    stack := TAIL(stack);
  ...............
    bx := bx + 1;
    CALL theloop;
    CALL endp
  END
  endp ==
    stack := < dx > ++ stack;
    CALL Z END
  ENDACTIONS
END ENDACTIONS;
```

Fig. 5. SumN – translated to an *Action system*

The Main Issues for Further Development. Although the initial results on small programs proved to be successful, there are several inherent issues with this approach that, when combined, make future development of the assembly tool less likely.

First is the question of feasibility of obtaining only high level structures in the translation (without auxiliary files). Second is the problem of assembly not being very standardised, even the order of the operands is not a sure thing, macros are defined in different ways, there are huge architectural changes, quite often there are "hacks" in the code, input output is done through hard to detect structures, etc. The authors were of course aware of these problem from the start, but were willing to sacrifice a good piece of the input domain in favour of higher quality end products.

Another problem is the lack of a good assembly base to experiment on. Both of the previous points make the selection of code very hard, and all of this is worsened by the lack of practical experience with assembly at our institution, which, taking into consideration the legacy aspects of the work, will probably not change in the future. This lack of "feel" for assembly and how programs are typically written with it is another negative factor.

The combination of these factors, as well as the options for the second tool that will be presented bellow, have led to the decision to focus the development of the tool and examples for use mainly in Software Evolution and potentially other courses.

```
program P
{
  void main()
  int i;
  {
    i = 0;
    while (i < 5) {
      print(i);
      i = i + 1;
    }
  }
}
```

```
14: enter 0 1
17: const_0
18: store_0
19: load_0
20: const_5
21: jge 13 (=34)
24: load_0
25: const_0
26: print
27: load_0
28: const_1
29: add
30: store_0
31: jmp -12 (=19)
34: exit
35: return
```

Fig. 6. MicroJava code and the translated bytecode

4. (MICRO)JAVA BYTECODE TRANSFORMATION

Java Bytecode is the language that is executed inside the standardised Java Virtual Machine[Lindholm et al. 2011]. In many ways it is similar to "classic" assembly languages, and therefore the reasons for translations and formal transformations apply here.

Most of the time bytecode is generated by a compiler from source code in the Java programming language, as it would be expected. But there are other languages and projects that try to use all the advantages of the standardised and very popular JVM that is available for most of the computer platforms that are in use today. For instance there are compilers for Python, Ruby, Pascal, C, Lisp, Scheme, PHP, JavaScript, and many other languages that produce Java Bytecode. The programming language Scala is compiled for either JVM or .NET machines.

The long term plan of this project is to build translators to and from WSL, that would allow both formal verification and transformation. While "regular" bytecode generated from Java can usually be decompiled quite successfully, this is not the case with code compiled from non-Java languages, or in cases where there was bytecode instruction injections for some particular purpose (such as persistance or additional security checks).

As a proof of concept the first step would be to work on a subset of the language. In this case the existing MicroJava specification was chosen.

MicroJava[2] was developed by Hanspeter Moessenboeck, for use in Compiler Construction courses with focus on the main features of a programming language without the distracting details[3]. For instance the only types are int and char primitives, arrays and basic class support. The concepts present in the MicroJava version of bytecode are very similar to "full" Java Bytecode, but simplified, with less instructions. It is also important to note that types are not explicitly encoded in MJ bytecode. An example of a program in this language and the code generated from it can be seen in Figure 6.

For the purposes of translating bytecode to WSL a new tool is being developed *mjc2wsl* (mjc – MicroJava Compiled). The basic concept are similar to asm2wsl – local variables are used to represent the registers, stack and other structures in the virtual machine. An example of translated bytecode can be seen in Figure 7.

The tool is currently in a closed prototype testing stage, but the plan is to publish it under an open source licence on the project's page[4].

[2]The name MicroJava may associate to "Java ME" (Micro Edition), but they are not related at all.
[3]Handouts for the course, available at http://ssw.jku.at/Misc/CC/
[4]http://perun.dmi.rs/pracner/transformations/

```
...                ACTIONS
                   .....
                   CALL a18 END
18: store_0        a18 ==
                   loc0 := HEAD(estack); estack := TAIL(estack);
                   CALL a19 END
19: load_0         a19 ==
                   estack := <loc0 > ++ estack;
                   CALL a20 END
20: const_5        a20 ==
                   estack := <5 > ++ estack;
                   CALL a21 END
21: jge 13 (=34)   a21 ==
                   tempa := HEAD(estack); estack := TAIL(estack);
                   tempb := HEAD(estack); estack := TAIL(estack);
                   IF tempb >= tempa THEN CALL a34 FI;
                   CALL a24 END
24: ....           .....
```

Fig. 7. MicroJava Bytecode and its WSL translation

5. CONCLUSIONS AND FUTURE WORK

This paper presents two tools for translating low level languages to a high level language *WSL*, which provides commands and structures for formal, provable transformations of the resulting code.

The first tool, *asm2wsl*, works with a subset of the Turbo Assembler flavor of x86 assembly language. The code gets translated into WSL with set up variables and structures that emulate the operations of the processor including all of the side effects. The complete logic of the program is translated to high level structures, without any memory mappings, unlike previous approaches[Ward et al. 2004]. Needless to say, the resulting code tends to grow in size, but this is not important as a series of automatic transformations can reduce the length of the program while keeping the logic intact. Manual transformations can be applied at any point for improved end results.

The initial results on small test programs were good, but this approach has a lot of limitations – many structures in assembly are practically impossible to detect and translate into high level counterparts. The input output system is a big problem for translation. Finally the available applicable code base and the amount of work being done at our institution with assembly is just not big enough for good tests. Therefore the decision was reached to turn this tool into a mainly educational helper in software evolution and potentially some other courses.

The second tool, *mjc2wsl*, works with MicroJava bytecode, which is a subset of Java bytecode (used in Java Virtual Machines). The basic inner workings are similar to the first tool. The strict specification solves most of the problems described above.

Direct bytecode changes can be used for a number of applications. *DYPER* [Reiss 2008] and *J-RAF2* [Hulaas and Binder 2008] are monitoring resources through instrumentation. *ASM Framework* [Kuleshov 2007] is used by a number of tools, including Jython, JRuby, Eclipse and some of Oracle's persistance systems. *Soot* is used in a number of research tools, as well as students' courses [Lam et al. 2011].

The end goal of this project is the expansion of *mjc2wsl* to Java bytecode and to take advantage of *WSL* that has already shown good results in real life applications and it's formally provable transformations, that are not available in other tools, both for optimisation and verification.

The obvious future steps are improvements to MJ bytecode automatic transformations, as well as developing translators from *WSL* back to bytecode and testing the potential improvements, and tools for verifying the correctness of the transformations. Further steps would be the expansion of the translation and transformation process to the complete Java Virtual Machine specification including two way translation between bytecode and *WSL*.

WSL has no static type system, and therefore transformations can not check type consistency which is a possible source of errors. For “full” Java Bytecode this would be necessary. A *Wide Spectrum Type System* was developed by Matthias Ladkau in his PhD thesis[Ladkau 2009], but it is not yet fully integrated into FermaT. This system could be used to improve many of the transformations, which is one of the goals of this project.

Another path of development would be the adaptation of the tools and good examples for courses in software evolution, software engineering and others.

ACKNOWLEDGMENTS

This work is partially supported by Ministry of Education and Science of the Republic of Serbia, through project no. OI174023: ”Intelligent techniques and their integration into wide-spectrum decision support”.

REFERENCES

Borland International 1990. *Turbo Assembler 2.0 User’s Guide*. Borland International.

Feng Chen, Hongji Yang, Bing Qiao, and William Cheng-Chung Chu. 2006. A Formal Model Driven Approach to Dependable Software Evolution. *Computer Software and Applications Conference, Annual International* 1 (2006), 205–214. DOI:http://dx.doi.org/doi.ieeecomputersociety.org/10.1109/COMPSAC.2006.10

Jarle Hulaas and Walter Binder. 2008. Program transformations for light-weight cpu accounting and control in the java virtual machine. *Higher-Order and Symbolic Computation* 21, 1-2 (2008), 119–146.

Eugene Kuleshov. 2007. Using the ASM framework to implement common Java bytecode transformation patterns. *Aspect-Oriented Software Development* (2007).

Matthias Ladkau. 2009. *A Wide Spectrum Type System for Transformation Theory*. Ph.D. Dissertation. De Montfort University, Leicester.

Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*. Galveston Island, TX.

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2011. *The Java Virtual Machine Specification, Java SE 7 Edition*. Oracle Ameria Inc.

Doni Pracner and Zoran Budimac. 2011. Understanding Old Assembly Code Using WSL. In *Proc. of the 14th International Multiconference on Information Society (IS 2011)*, Vol. A. Ljubljana, Slovenia, 171–174.

Steven P Reiss. 2008. Controlled dynamic performance analysis. In *Proceedings of the 7th international workshop on Software and performance*. ACM, 43–54.

Harry Sneed. 2000. Encapsulation of legacy software: A technique for reusing legacy software components. *Annals of Software Engineering* 9 (2000), 293–313. Issue 1. http://dx.doi.org/10.1023/A:1018989111417 10.1023/A:1018989111417.

Martin Ward. 1989. *Proving Program Refinements and Transformations*. Ph.D. Dissertation. Oxford University.

Martin Ward. 1999. Assembler to C Migration using the FermaT Transformation System. In *IEEE International Conference on Software Maintenance (ICSM’99)*. IEEE Computer Society Press, 67–76.

Martin Ward. 2000. Reverse Engineering from Assembler to Formal Specifications via Program Transformations. In *7th Working Conference on Reverse Engineering, Brisbane, Queensland, Australia*. IEEE Computer Society.

Martin Ward. 2004. Pigs from Sausages? Reengineering from Assembler to C via FermaT Transformations. *Science of Computer Programming, Special Issue on Program Transformation* 52/1-3 (2004), 213–255. DOI:http://dx.doi.org/dx.doi.org/10.1016/j.scico.2004.03.007

Martin Ward and Hussein Zedan. 2005. METAWSL and Meta-Transformations in the FermaT Transformation System. In *IN COMPSAC*. IEEE Computer Society, 233–238.

Martin Ward, Hussein Zedan, and Tim Hardcastle. 2004. Legacy Assembler Reengineering and Migration. In *ICSM2004, The 20th IEEE International Conference on Software Maintenance*. IEEE Computer Society.

E.J. Younger, K.H. Bennett, and Z. Luo. 1997. A Formal Transformation and Refinement Method for Concurrent Programs. *Software Maintenance, IEEE International Conference on* 0 (1997), 287. DOI:http://dx.doi.org/doi.ieeecomputersociety.org/10.1109/ICSM.1997.624256

Documenting Software Architectures using UML

FILIP PRENTOVIĆ, GTECH, Belgrade
ZORAN BUDIMAC, University of Novi Sad

As software systems become large and more complex, focus on main design issues is shifted from algorithms and data structures. Software architecture, which represents high-level organization of software system, brings whole new set of design issues: overall system organization, global control structures, communication protocols, data access and synchronization, as well as choosing between different design solutions. In this paper, ways of using UML to document component and connector views are described. Following elements of component and connector view will be described using UML: components and component types, connectors and connector types, ports, roles, systems and properties.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures

General Terms: software architecture, UML

Additional Key Words and Phrases: architecture description languages, component and connector view

1. INTRODUCTION

As software systems become large and more complex, focus on main design issues is shifted from algorithms and data structures. Software architecture, which represents high-level organization of software system, brings whole new set o[i]f design issues: overall system organization, global control structures, communication protocols, data access and synchronization, as well as choosing between different design solutions.

Although architectural descriptions of software systems play important role in software design, they are often represented in informal way, by using simple diagrams with ad-hoc notation, with little or no influence on later phases of software development. To overcome this, a number of architecture description languages (ADLs) have been developed, in order to improve understandability, reusability and analysis capabilities of architectural descriptions. Also, ADLs are formal way of representing architectures, they permit analysis and assessment of architectures, for completeness, consistency, ambiguity, and performance, and, in some cases, can support automatic generation of software systems [Clements et al., 2002].

Despite aforementioned advantages of ADLs, there are several disadvantages of using ADLs, which limited their use in practice. First and foremost, there is lack of support by commercial tools, primarily because there's no universal agreement on what ADLs should represent. Because of this, UML was proposed as a solution for describing software architectures. UML has many clear advantages: large number of commercial tools, widespread use, as well as implementation capabilities.

In this paper, possible strategies for representing component and connector view in UML are presented- Section 2. Section 2.1 contains possible ways of describing architectural components, section 2.2 presents strategies for description of ports, section 2.3 depicts some possible solutions for describing architectural connectors using UML concepts, while strategies for describing systems and properties are presented by sections 2.4 and 2.5, respectively. Related work is provided by section 3. Conclusion and future work are given by section 4. Glossary of terms used in this paper is given by section 5.

2. DOCUMENTING COMPONENT AND CONNECTOR VIEWS

In this section, ways of using UML (class diagram in particular) to document component and connector view are described. Following elements of component and connector view will be described using UML:

Author's address: F. Prentović, Bul. M. Pupina, 11000 Belgrade, Serbia; email: filipprentovic@yahoo.com; Z. Budimac, Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia; email: zjb@dmi.uns.ac.rs.

In: Z. Budimac (ed.): Proceedings of the 2nd Workshop of Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA), Novi Sad, Serbia, 15.-17.9.2013, published at http://ceur-ws.org

components and component types, connectors and connector types, ports, roles, systems and properties [Garlan et al., 2002].

Since there is no best way of documenting these elements using UML, multiple alternatives will be presented, along with their advantages and disadvantages. Architectural descriptions which are produced this way should respect documented UML semantics and the intuitions of UML modelers. The interpretation of the encoding UML model should be close to the interpretation of the original architectural description so the model is intelligible to both designers and UML-based tools. Also, resulting architectural descriptions in UML should bring conceptual clarity to a system design, avoid visual clutter, and highlight key design details. All relevant architectural features for the design should be represented in the UML model, keeping in mind that not all uses of UML are supported equally by all UML tools, which can limit documentation options [Clements et al., 2002].

It's worth pointing out that strategies for documenting component and connector view in UML presented in this paper rely heavily on concepts introduced in UML 2.0, such as structured classifiers and composite structure diagrams. A composite structure diagram depicts the internal structure of structured classifiers (e.g. classes and components) by using parts, ports and connectors, and some of these concepts appear as a natural candidate for appropriate architectural elements. For the explaining purposes, simple publish-subscribe architecture, in which publisher and subscribers communicate through the event dispatcher (represented as event "bus"), will be considered (Figure 1) [Clements et al., 2002].

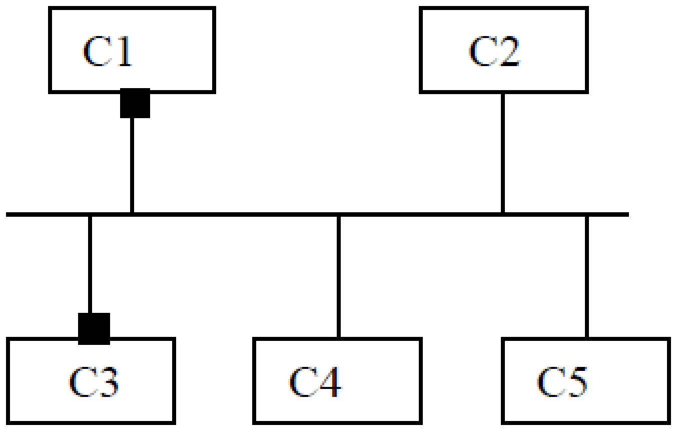


Figure 1. Simple publish-subscribe architecture

2.1 Documenting components

There are two meaningful ways of representing architectural components in UML, by using UML classes or UML components.

A natural candidate for representing component types in UML is the class concept. Classes describe the conceptual vocabulary of a system just as component types form the conceptual vocabulary of architectural documentation. Also, UML classes, like component types in architectural descriptions, are first-class entities and rich structures for capturing software abstractions.

Structural properties of architectural components can be represented as class attributes or associations, behavior can be described using UML behavioral descriptions, and generalization can be used to relate a set of component types. The type/instance relationship in architectural descriptions is a close match to the class/object relationship in a UML model, although it's not identical. A component instance might refine the number of ports specified by its type or associate an implementation in the form of an additional structure that is not part of its type's definition. This can be overcome by using subclasses, so that instance of a subclass represents instance of a component.

On the other hand, UML components have an expressive power similar to that of classes and can be used to represent architectural components as in the first strategy, with slightly different graphical notation. In addition, component can own packageable elements along with elements a class can own, which can be useful in some cases.

Choosing between these two strategies may rely more on the semantic match offered by them, since both have nearly the same expressiveness, and are visually nearly identical. One thing which can influence the decision is a way in which connectors are documented. For example, classes can be used to describe connectors, in which case using classes for describing components can have negative impact on visual clarity of architectural description.

2.2 Documenting ports

Most natural way of representing architectural ports in UML is using port concept, introduced along with classifier concept. UML ports are explicit interaction points for classifiers, so it can be used with both classes and components. UML ports provide all the expressiveness needed to document architectural ports. Multiple ports can be defined for a component type, enabling different interactions. Since ports in UML can be typed by an interface or a class, multiple instances of the same port type are allowed. Ports can also be associated with multiple interfaces, provided and required (Figure 2).

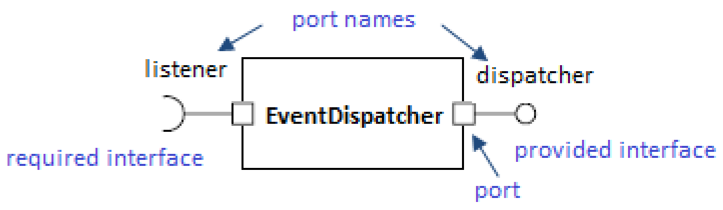


Figure 2. UML class with ports

Ports can be omitted in some diagrams, in case of single port components, or if the ports can be inferred from the system topology. Identifying the ports of a component allows different interactions to be distinguished based on the port through which they occur.

2.3 Documenting connectors

Even though there is connector concept in UML, it lacks expressiveness needed to be considered as a solution for documenting architectural connectors, e.g. it lacks ability to associate semantic information with a connector or to clearly document architectural connector roles. Therefore, three strategies for representing connectors in UML will be considered:

- ☐ using UML associations
- ☐ using UML association classes
- ☐ using UML classes

Using UML associations for documenting connectors allows visual distinction between components and connectors. Different types of connectors can be distinguished by labeling each association with a stereotype (Figure 3).

This strategy lacks ability to express semantic information connectors provide, because roles of connector can't be defined by associations (since associations don't have UML interfaces or ports), and associations can't own neither attributes nor behavioral descriptions. Nevertheless, this strategy is useful when purpose of documentation is identifying where different types of connectors are used in a system.

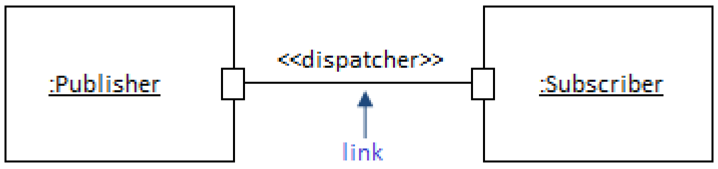


Figure 3. Documenting connector using link

The second strategy consists of using UML associations class for describing connectors (Figure 4). This strategy allows semantic descriptions of connectors, by using attributes and behavioral descriptions of a class, as well as creating substructure of connectors, if needed.

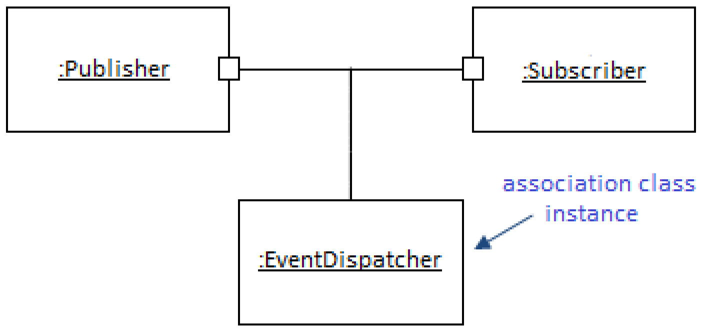


Figure 4. Documenting connector using association class instance

Also, connector roles now can be represented as UML ports on the association class, which enables same level of expressiveness that component ports have (Figure 5).

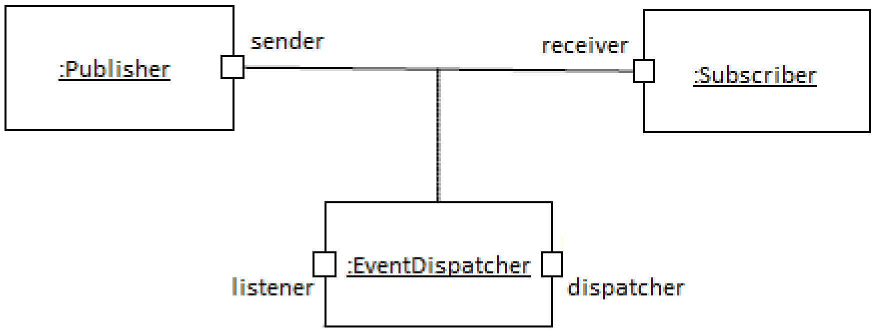


Figure 5. Documenting connector using association class instance with ports

However, this brings out issue of attaching component ports to connector roles. This can be solved either by role name being added to corresponding component port, or by using assembly connectors between ports of objects representing the connector and the ports of the objects representing the components. First solution doesn't affect visual clarity, but it lacks standard tool support, and second solution introduces substantial visual clutter.

The third strategy consists of using UML class for documenting connectors (Figure 6).

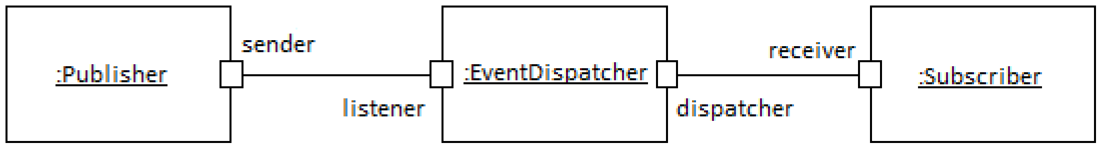


Figure 6. Documenting connector using object

This strategy allows same expression capabilities as the previous one, but also resolves the component and connector attachment problems by explicitly using UML assembly connectors, removing the potential ambiguity of the second strategy. Unfortunately, this solution presents the poorest visual distinction between components and connectors, especially if classes are used for documenting components. This problem can be mitigated by using different UML concepts to represent components and connectors, as shown in Figure 7, where UML components are used for representing architectural components, and class is used for representing connector.

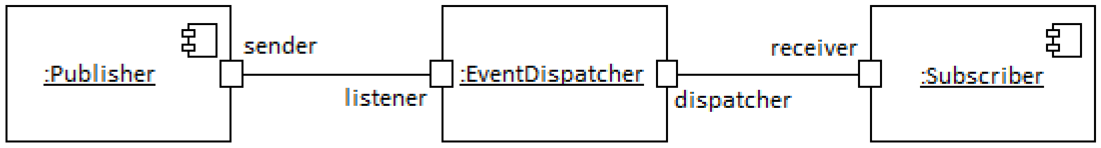


Figure 7. Documenting connector using object, UML components document components

However, this variation is only slightly better in terms of visual distinctiveness. Another solution to this problem would be using UML stereotype mechanism, which allows customized visualization of stereotyped elements (Figure 8). However, this solution has limited practical application, since use of stereotypes requires graphical support not offered by most UML tools.

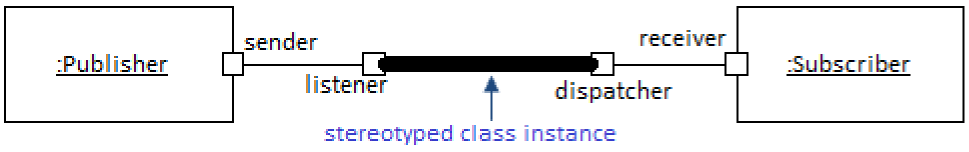


Figure 8. Documenting connector using stereotyped class

Decision on which strategy is the most suitable for documenting connectors should be made depending on answers for following questions:

- ☐ Does the architectural description need only types of connector being identified, and/or effects of connector on component interaction?
- ☐ For which phase of system's development lifecycle architectural description is being written?
- ☐ Which strategy is being used for documenting components?
- ☐ What tool support is available?

If the goal of documentation is to identify where different types of connectors are used in a system, particularly if the connector types are well known and understood, first strategy is a good choice. This strategy is a good choice for "first drafts", in which specific connector semantics have not been defined, but crude choices should be identified by name. Drawback of this strategy is its inability to describe semantics and role of a connector. Second strategy is a good choice when connector semantics need to be described, but specific component port and connector role attachments are not important. If these attachments are important, third strategy can be used.

2.4 Documenting systems

Documenting systems relies on definitions of component and connector types, as well as topologies of instances of component and connector types. Types and their hierarchies are documented using class and component diagrams. For subtypes generalizations can be used, and for types decomposition classes can be used. Topologies of component and connector instances can be documented using instance diagrams.

2.5 Documenting properties

Architectural properties capture semantic information about a system. Therefore, UML properties concept can be used for describing them, because UML properties represent structural feature. There are three possible ways of documenting properties in UML:

- ☐ using tagged values
- ☐ using attributes
- ☐ using stereotypes

Tagged value is name-value pair that can be used for documenting information semantically relevant to classifier, and therefore can be used to document architectural properties. Downsides of this approach are that there is no explicit documentation of the value's type, and that tagged value can be defined only for instance.

UML attributes can't be used directly for describing properties because they represent structural elements. This can be overcome by using stereotype denoting that an attribute is semantic, not structural.

Third strategy consists of using stereotypes as a way of extending UML meta model and allow new semantics to be incorporated into UML models. One way to extend a concept is to define a stereotype that includes a tag definition. When the stereotype is applied, the value of a tagged definition is called a tagged value.

Each of these strategies has advantages over the others. If analysis tools are not used and properties are not required for all instances of a class, the first strategy is adequate. The second strategy is a good choice if the properties are mandatory to support analysis, but the implementation consequences are not terribly detrimental. The third strategy also provides explicit documentation of the property type, but lacks the semantic mismatch and potential implementation consequences of the second strategy.

3. RELATED WORK

Concepts introduced in UML 2 are used in [Anacleto 2008], where UML profile and a group of UML patterns for documenting the component and connector view of software architectures are presented. In this work, UML components are used for representing connectors, with stereotypes used for visual distinction from architectural components represented by UML components as well. A different approach is considered in [H. B. Christensen et al., 2011], with component instances represented as UML objects, and connector instances represented as links. Some aspects of hierarchical decomposition of a system into modules in UML 2 are discussed in [Frick et al., 2004].

Apart from strategies for documenting component and connector view described previously, attempts were made in order to describe different aspects of software architectures using concepts that existed in UML 1.x. Using UML 1.x in documenting different architectural views is considered in [Hofmeister et al., 1999]. Although the conclusion was that UML works well for describing a static structure of the architecture, it also emphasized its lack of support for describing ports on components, among other things. Constraining and extending UML 1.x meta model is considered in [Medvidovic et al., 2002], in addition to using existing UML notation. Constraining UML meta model seems appropriate in trying to enforce architectural constraints, while augmenting meta model could be beneficial in trying to incorporate new modeling capabilities. Component-modeling capabilities of UML are presented in [Kobryn 2000], with the emphasis on modeling component frameworks like EJB and COM+, which shows that UML 1.3 provides basic support for this type of modeling, with issues that comes from semantics overlap between components and classes, as well as lack of support for large component systems and frameworks.

4. CONCLUSION AND FUTURE WORK

In this paper, multiple strategies have been presented for documenting elements of architectural descriptions in UML, along with guidelines for determining which strategy represents the best solution. It is obvious that UML 2 is more suitable for describing component and connector view of software architecture than its predecessors. However, there are a few issues that continue to make documenting architectures relatively complex: UML connectors are not first class entities, so less natural representations of architectural connectors must be used, and there isn't a completely natural way of documenting architectural properties. Also, concept of a part, introduced with composite structure diagrams in UML 2 can be taken into consideration when showing composite properties of the containing structured classifier. Furthermore, documenting other views of software architectures in UML can be considered, along with more complex and more illustrative examples.

5. GLOSSARY

Since software architecture has many definitions, and since architectural descriptions differ depending on a phase of software development life-cycle in which they are used, a glossary is needed in order to avoid confusion, and also to explain terms with potentially ambiguous meaning. Here is the list with short definition of terms which are used throughout the paper:

- ☐ software architecture - structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them [Clements et al., 2002]
- ☐ architecture description languages (ADLs) - language and/or a conceptual model to describe and represent system architectures
- ☐ component and connector view - define models consisting of elements that have some run-time presence, such as processes, objects, clients, servers, and data stores. Additionally, component and

connector models include as elements the pathways of interaction, such as communication links and protocols, information flows, and access to shared storage [Clements et al., 2002].

- ☐ components and component types - represent the principle runtime elements of computation and data storage such as clients, servers, filters, and databases
- ☐ connectors and connector types - represent the runtime pathways of interaction between components such as pipes, publish-subscribe buses, and client-server channels
- ☐ component interfaces (or ports) - represent points of interaction between a component and its environment
- ☐ connector interfaces (or roles) - represent points of interaction between a connector and the components to which it is connected
- ☐ systems - graphs representing the components and connectors in the system and the pathways of interaction among them
- ☐ properties - additional information associated with structural elements of an architecture
- ☐ styles - define a vocabulary of component and connector types together with rules for how instances of those types can be combined to form an architecture in a given style e.g. pipe-and-filter, client-server, and publish-subscribe

REFERENCES

L. V. A. Anacleto, A UML Profile for Documenting the Component-and-Connector Views of Software Architectures. Epidata Consulting, April 2008.

P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, Documenting Software Architectures: Views and Beyond. Boston, MA: Addison-Wesley, 2002.

H. B. Christensen, A. Corry and K. M. Hansen, The 3+1 Approach to Software Architecture Description Using UML, Department of Computer Science, University of Aarhus, 2011

G. Frick, B. Scherrer and K. D. Muller-Glaser, Designing the Software Architecture of an Embedded System with UML 2.0. Software Architecture Description & UML Workshop, October 2004.

D. Garlan, S. Cheng and A. J. Kompanek, Reconciling the Needs of Architectural Description with Object Modeling Notations. Science of Computer Programming, Special UML Edition 44, 1 (July 2002): 23-49.

C. Hofmeister, R. L. Nord and D. Soni, Describing Software Architecture with UML. First Working IFIP Conference on Software Architecture, February 1999.

C. Kobryn, Modeling Components and Frameworks with UML. Communications of the ACM, vol. 43, no. 10, October 2000.

N. Medvidovic, D. Rosenblum, D. Redmiles and J.Robbins, Modeling Software Architecture in the Unified Modeling Language. ACM Transactions on Software Engineering and Methodology (TOSEM) 11, 1 (January 2002): 2-57.

Redefining Software Quality Metrics to XML Schema Needs

MAJA PUŠNIK, BOŠTJAN ŠUMAK AND MARJAN HERIČKO, University of Maribor
ZORAN BUDIMAC, University of Novi Sad

The structure and content of XML schemas, important and widely used document definitions, has a significant influence on the quality of XML data and XML technologies in general, therefore the quality of XML Schemas and accurate assessment of the quality is a fundamental research challenge in all fields of XML application. A good quality estimation of an XML schema can directly and indirectly lead to a higher efficiency of its usage, simplification of information solutions, efficient maintenance, and higher quality of data and business processes. This paper addresses challenges in measuring the level of XML schema quality by employing general software quality metrics; a set of holistically defined and document-oriented metrics is proposed. Proposed XML Schema quality metrics base on existing software metrics, adapted according to needs of XML schemas, addressing it mostly from a structural perspective.

Categories and Subject Descriptors: **H.0. [Information Systems]**: General; **D.2.8 [Software Engineering]**: Metrics — *Complexity measures; Product metrics;* **D.2.9. [Software Engineering]:** Management — *Software quality assurance (SQA)*

General Terms: Software quality assurance

Additional Key Words and Phrases: software metrics, quality metrics, XML Schema

1. INTRODUCTION

The primary role of XML schemas is the definition of XML data and supporting rules regarding the use of XML data, an important part of information technologies. XML schemas and related technologies present an important part of IT solutions in most Slovenian companies [Sušnik 2008], EU and the world [Rishel 2011]. Using XML has spread from the field of e-business and data exchange to data presentation into various levels of contemporary information solution architectures: (1) web service interface definitions, (2) data models, (3) specification of business cooperation protocols between different companies (their many uses are evident from different scientific and technical papers), etc.. Due to the widespread use, the question of XML schema quality is often open, particularly from the aspect of structure (and content) of XML schemas, which indirectly influence the quality of data that XML schema describes. Therefore measuring XML schemas quality is the basic research challenge in our paper. Solution of the problem (the composite of metrics) will directly or indirectly lead to greater efficiency in the use of XML schemes, simplifying IT solutions, facilitating maintenance, improving the quality of data and associated business processes. Ideally the metrics should apply the aspect of structure, content and domain, in which the XML schema is applied, however this paper will focus mostly on structural aspect, trying to take advantage of existing software metrics.

There have been several attempts to evaluate and measure XML schemas. Few of them are summed in [Zhang 2008]. Significantly related work was also done in [McDowell, Schmidt, Yue 2004] and [Narasimhan, Hendradjaya 2007], where attempts to measure XML schemas as well as software in general were made. The subject was addressed in other papers, not included in this overview, however the background are mainly software metrics, which do not necessary always apply needs of XML schema quality (and complexity) measurements.

Based on surveys and interviews, conducted within the University of Maribor and nearby companies, XML Schemas are often built irrationally in a manner, which satisfies the minimum requirements of syntactic correctness and content sufficiency. Existing metrics only partially address the problem basing

Author's address: M. Pušnik, B. Šumak, M. Heričko, Institute of informatics, Faculty of Electrical Engineering and Computer Science, Smetanova ulica 17, 2000 Maribor, Slovenia, email: {maja.pusnik, bostjan.sumak, marjan.hericko}@uni-mb.si; Z. Budimac, Department of mathematics and informatics, Faculty of Sciences, University of Novi Sad, Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia, email: zjb@dmi.uns.ac.rs

In: Z. Budimac (ed.): Proceedings of the 2nd Workshop of Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA), Novi Sad, Serbia, 15.-17.9.2013, published at http://ceur-ws.org

on existing solutions known in software engineering and not addressing the problem of an objective quality evaluation of an XML Schema. Dynamic creation and adaptation of XML schemas schedules and presents an additional research challenge that requires the use of new approaches and solutions, universal and specific according to a domain.

The aim of this paper is definition of a new theoretical approach for evaluating the quality of XML Schema, basing on the original concept of semantically related analysis of XML schemes and XML documents, by using a new set of metrics. The design correctness of the newly redefined metrics was confirmed on an expanded set of test data of already established XML schemes in the field of e-business and integration of complex business information systems. For quality measurement purposes we gathered quality parameters, addressing different aspects of XML Schema needs and demands.

This paper is organized into four chapters. After the presentation of this papers background and the description of included XML quality parameters, chapter two presents all aspects in metric types. Chapter 3 presents metric application and chapter four includes discussion of our present work and future plans.

1.1 XML schema quality parameters

The results of a systematic review of literature in the field of measuring XML schemas showed that several metrics were applied to XML schema evaluation, extracted mainly from the methods of software engineering measurements, focusing mostly on the complexity of XML Schemas. To include a variety of parameters addressing complexity and quality, we searched different fields on quality measurement. The first group of parameters was related to the structural characteristics of XML schemes (we included a survey, where all currently defined metrics are taken from several authors in [Zhang 2008]):

- *XML schema size,*
- *Number of XML nodes and annotations,*
- *Number of global and local element declarations,*
- *Number of global or local complex types definitions,*
- *Number of derived complex types, number of global and local definitions of simple types,*
- *Number of global or local definitions of models groups (groups),*
- *Number of global or local definitions of groups of attributes,*
- *Branch elements, the average cardinality of elements, etc.*

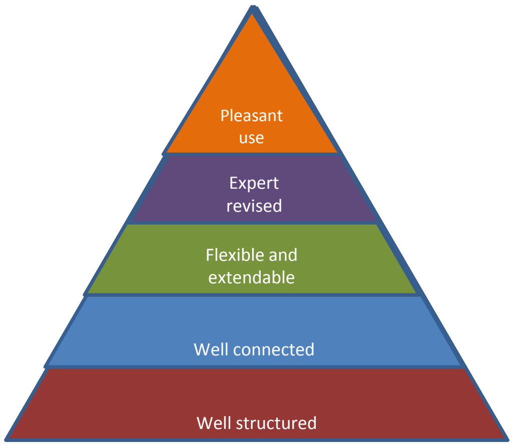


Fig. 1 Quality hierarchy in XML schemas

The typically software metrics parameters were extended with parameters form other quality measurement fields, specifically taken from standards ISO (ISO/IEC 9126 [McDowell, Schmidt, Yue 2004]), decision models theory [Burris 2012] and other papers [Zhang 2008]):

- *XML schemas functionality*
- *XML schemas simplicity*
- *XML schemas scalability*
- *XML schemas comprehensibility*
- *XML schemas re-use,*

- *XML schemas fullness,*
- *XML schemas integrability,*
- *XML schemas Flexibility,*
- *XML schemas Implementation,*
- *XML schemas Maintenance,*
- *Accuracy,*
- *Validity,*
- *Up to date,*
- *Minimalism,*
- *Consistency,*
- *Portability*
- *Security,*
- *Interoperability*
- *Reliability,*
- *Effectiveness,*
- *Visibility*

To determine the quality levels of XML schema usage, we borrowed Maslow's hierarchical nature needs, which can be applied to software and to all supporting technologies, presenting our interpretation in Fig. 1. The gathered parameters were organized into six groups, reflecting six identified XML schema needs respectively XML schema quality demands, meeting the three main XML schema demands: (1) good structure, (2) consistent contents, (3) compliant with domain. All parameters, contributing to XML schema quality and all aspects of quality are combined in Fig. 2.

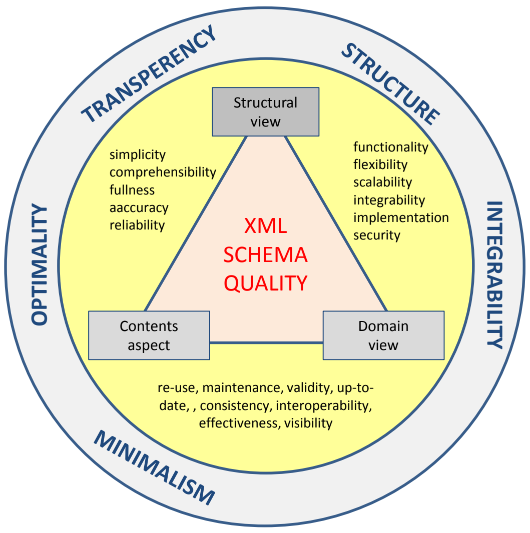


Fig. 2 Quality aspects in XML schemas

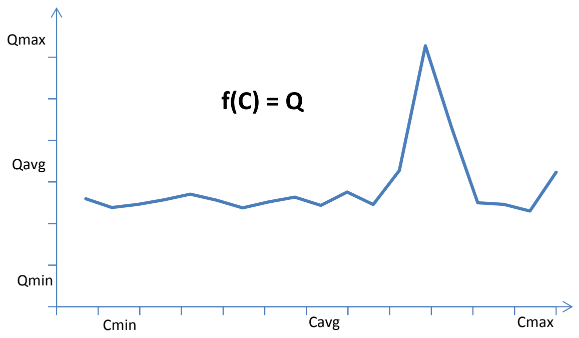


Fig. 3 Quality-complexity dependance

2. METRIC TYPES

So that individual metrics could be compared, NORMALIZATION of parameters was conducted. All the parameters that were used within the metrics and their results were transformed to a scale of 0 to 1, where 0 represented the worst value for each parameter and 1 the best value. The transformations based on linear programming, assuming that the growth relationship is linear. The following metrics address all aspects of XML schema quality.

2.1 Structural aspect

Other authors have researched measuring the structure of XML schemes for calculating the complexity and quality by McDowell and others [Burris 2012]. The authors present a number of metrics, taken mainly from "quality model" ISO standard and link them into a single formula. Each variable is further multiplied, however the factors are not justified, values are not normalized, so the formula cannot be applied, but we have analysed and partly used in our calculation formula of quality.

Within the complexity calculations we can conclude that the higher the value of the individual, the greater the complexity (the relationship is shown in Fig. 3). According to XML schema needs we redefined metrics into the following composite metric (1) with the following parameters:

- *S1 - relationship between simple and complex data types*
- *S2 - relationship between annotations and the number of elements*
- *S3 - average number of restrictions on the declaration of a simple type*
- *S4 - percentage of the derived type declarations of total number of declarations complex types*
- *S5 - diversification of the elements or 'fanning' which is influenced by the complexity of XML schemas suggesting inconsistencies in XML schemas that unnecessarily increase the complexity*

$$Q1 = \frac{S1 + S2 + S3 + S4 + S5}{5} \qquad (1)$$

2.2 Transparency and documentation of the XML Schema

The importance of well documented and easy-to-read/understand XML schema is addressed in the following relationship: number of annotation (N_{An}) depending on the number of items (N_E) and attributes (N_{At}) illustrates the documentation of XML schemas, supposing that more information about the building blocks increases the quality. The parameters in metric 2 regard transparency and documentation.

$$Q2 = \frac{N_{An}}{N_E + N_{At}} \qquad (2)$$

2.3 XML schema optimality

In metric 3 we combined several parameters, indicating the optimal structure of an XML Schema. The metric evaluates whether the in-lining pattern has been used, the least preferable one in XML schema building. In doing so, we focus on the following relationships:

- *(O1) The relationship between local and all elements*
- *(O2) The relationship between local attributes and all attributes*
- *(O3) The relationship between global and complex elements of all the complex elements*
- *(O4) The relationship between global and all the simple elements of simple elements.*

Ratio between XML schema building blocks (O1, O2, and O4) should be minimized; meaning minimisation of local elements and attributes and more global simple and complex types; the number of global elements (O3) should be as low as possible, due to the problem of several roots (such flexibility is not always appreciated). This particular parameter differentiates domains into two groups (the flexible ones appropriate to validate multiple different XML schemas, and the strict ones, striving to one root policy for validity or other reasons). In metric 3 we assumed the majority of XML schemas want a certain level of flexibility, therefore the aspect of security was disregarded.

$$Q3 = \frac{O1 + O2 + (1 - O3) + O4}{4} \qquad (3)$$

The metrics, described in the following subchapters, use a similar set of parameters:

- *(N_E) Number of elements*
- *(N_{At}) Number of attributes*
- *(N_{An}) Number of annotations*
- *(LOC) Number of lines of code*
- *(N_{re_all}) - number of references to elements (simple and complex)*

- *(N_{ra_all}) - number of references to attributes*
- *(N_{rg_all}) - number of references to groups (elements and attributes)*
- *(N_{ri_all}) - the number of schemes and imported*
- *(Ng) - The number of groups*

2.4 XML schema minimalism

In this metric we combine the parameters that indicate the minimum XML schemas building blocks, where the concept of minimalism is defined as the level, where one can anticipate that there is no other set of less building blocks, however still descriptive full:

$$Q4 = \frac{N_{An} + N_E + N_{At}}{LOC} \tag{4}$$

2.5 XML schema re use

The equation was inspired by author [Washizaki, Fukazawab 2005], where we summed up and defined a set of metrics for measuring the re-use of the software. The metric includes parameters that allow the re-use and are inherently global. We included the following parameters:

$$Q5 = \frac{N_{re_{all}} + N_{ra\_all} + N_{rg\_all} + N_{ri\_all}}{N_E + N_{At} + N_g} \tag{5}$$

2.6 XML schema integrability

Definition of equation was taken from the idea of density of software components [Narasimhan 2007], where the authors calculate the density of the other segments of the software and the density of interactions between them (lines of code, operations, classes, modules ...).We adjusted and simplified the formula into the following equation:

$$Q6 = \frac{N_E + N_{At} + N_g + N_{re_{all}} + N_{ra\_all} + N_{rg\_all} + N_{ri\_all} + N_{re_{all}} + N_{An}}{N_E + N_{At}} \tag{6}$$

3. METRICS APPLICATION

We tested proposed metrics on a set of 200 XML schemas, subtracted from different domains, acknowledging several standards, available on the market in a certain domain. Each XML schema was evaluated manually and automatically with proposed metrics, eliminating possible duplicates due to crossing of different fields. The results of all metrics were combined and nominated to a scale from 1-3, where a level 1 schema is of high quality and level 3 XML schema is of low quality (using identical scale in case of the manual evaluation). Comparing the two types of evaluation, 83% of data received an equal evaluation (Fig. 4).

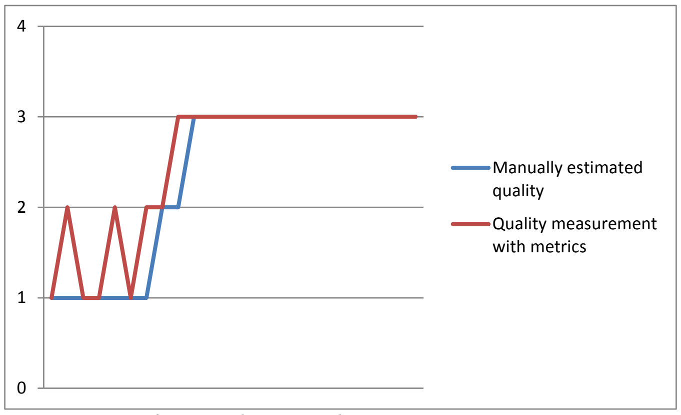
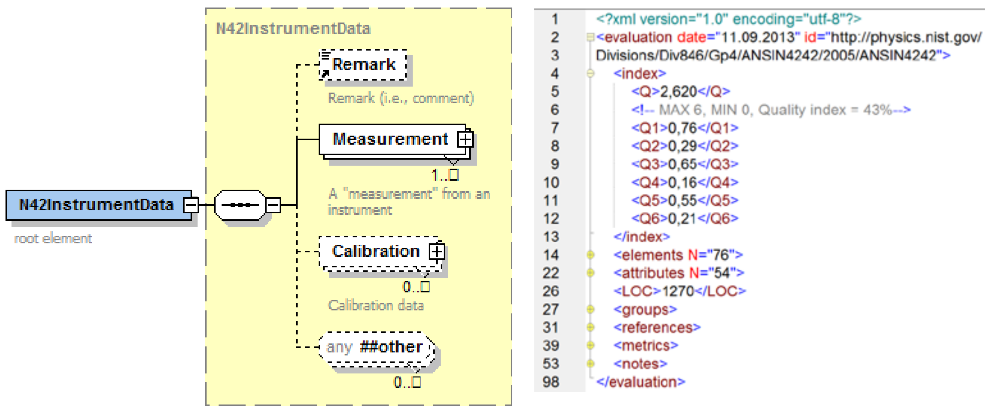


Fig. 4 Manual and metrical measurement of XML schema quality.

All metrics were considered as equal, therefore no priority weights are applied to each metric. This limitation was used due to simplification of our early stage metric framework; weights were omitted for the length purposes, since the paper does not include domain/aspect priorities clarification. We treated all aspects of XML schema as equal due to heterogeneous domain, which were not explored in this paper. Definition of weights will be a part of our future work. For the purposes of this paper, we used the following equation:

$$Q = Q_1 + Q_2 + Q_3 + Q_4 + Q_5 + Q_6 \tag{7}$$

A presentation of metrics application is shown in figure (Fig. 5).A sum of 220 real-life standard or semi-standard XML schemas was used to apply defined metrics. Evaluation software produced a resulting XML document with a summary of all data, some warnings or eventual errors and metric results.



```
<?xml version="1.0" encoding="utf-8"?>
<evaluation date="11.09.2013" id="http://physics.nist.gov/
Divisions/Div846/Gp4/ANSIN4242/2005/ANSIN4242">
   <index>
      <Q>2,620</Q>
      <!-- MAX 6, MIN 0, Quality index = 43%-->
      <Q1>0,76</Q1>
      <Q2>0,29</Q2>
      <Q3>0,65</Q3>
      <Q4>0,16</Q4>
      <Q5>0,55</Q5>
      <Q6>0,21</Q6>
   </index>
   <elements N="76">
   <attributes N="54">
   <LOC>1270</LOC>
   <groups>
   <references>
   <metrics>
   <notes>
</evaluation>
```

Fig. 5 Metric application example based on an XML schema.

4. DISCUSSION

The focus of the paper was definition of a full set of parameters for assessing the quality of XML schemes, trying to include all aspects and needs of XML schema quality. We defined six metrics, focusing on important aspects of XML schema quality, and repositioned XML schema facts into parameters, measuring the importance of each building block. To assure correctness, we evaluated each XML schema manually based on a simple overview, noting clearness and readability; and compared our results with metrics' results. The overlapping was at 83%.

Correct (and quick) measurement of XML Schema quality provides a strategic decision-making and improvement in data organization, as a standard mechanism (internal or global) for evaluation of XML schemes quality. Software metrics are a good basis for XML schema quality measuring, however some accommodations are necessary according to their needs and demands. As users operate with different data from multiple domains of XML technologies application, the quality measurements vary depending on the flexibility (or inflexibility) of structures.

In future work we will further explore applicability of defined metrics, their success and validity on practical examples and the need for metrics adaptability according to the domain in which an XML schema is used.

REFERENCES

Zhang, Y. (2008). *Literature Review and Survey: XML Schema Metrics*.

Wes Rishel. (2011). *Does XML Schema Earn its Keep?* The Gartner Blog Network. http://blogs.gartner.com/wes_rishel/2011/12/31/ok-xml-schema-does-earn-its-keep-in-hl7/

Sušnik, M. (2008). *V slogi je e-račun! Monitr Pro*, http://www.monitorpro.si/41040/praksa/v-slogi-je-e-racun/.

Standard ISO/IEC 9126 Software engineering

McDowell, A., Schmidt, C., Yue, K. (2004). *Analysis and Metrics of XML Schema*. Proceedings of the International Conference on Software Engineering Research and Practice, SERP'04, v 2, p 538-544, 2004.

Burris, E. (2012), Hierarchical Nature of Software Quality, Programming in the Large, The Practice of Software Engineering, http://programminglarge.com/hierarchical-nature-of-software-quality/.

Narasimhan, V.L., Hendradjaya, B. (2007). *Some theoretical considerations for a suite of metrics for the integration of software components*. Information Sciences, Volume 177, Issue 3, 1 February 2007, Pages 844-864. http://dx.doi.org/10.1016/j.ins.2006.07.010

Washizaki, H., Fukazawab, Y. (2005). *A technique for automatic component extraction from object-oriented programs by refactoring*. Volume 56, Issues 1–2, April 2005, Pages 99–116. http://dx.doi.org/10.1016/j.scico.2004.11.007

SSQSA Ontology Metrics Front-End

MILOŠ SAVIĆ, ZORAN BUDIMAC, GORDANA RAKIĆ AND MIRJANA IVANOVIĆ, University of Novi Sad
MARJAN HERIČKO, University of Maribor

SSQSA is a set of language independent tools whose main purpose is to analyze source code of software systems in order to evaluate their quality attributes. The aim of this paper is to present how a formal language that is not a programming language can be integrated into the front-end of the SSQSA framework. Namely, it is explained how the SSQSA front-end is extended to support OWL2 which is a domain-specific language for the description of ontological systems. Such extension of the SSQSA front-end represents a step towards the realization of a SSQSA back-end which will be able to compute a hybrid set of metrics that reflect different aspects of complexity of ontological descriptions.

Categories and Subject Descriptors: D.2.8 [**Software Engineering**]: Metrics – *Complexity measures*; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods – *Representation languages*

General Terms: Languages, Measurement

Additional Key Words and Phrases: OWL2, Ontology metrics, Complexity, SSQSA, eCST representation

1. INTRODUCTION

With the rise of the semantic web, ontologies have become a key technology to provide formal description of shared and reusable knowledge. Viewed as "explicit specification of conceptualization" [Gruber 1993], ontologies are used to define concepts and relations present in a domain in order to support reasoning, integration, and aggregation of data by autonomous software agents. Since real-world ontologies rapidly increase in size, it has become highly important to measure, evaluate and understand their complexity, in order to be able to control their maintenance and evolution.

SSQSA is a set of language-independent tools that statically analyze software systems in order to evaluate their quality attributes [Budimac et al. 2012]. The whole framework is organized around the enriched Concrete Syntax Tree (eCST) representation of source code [Rakić and Budimac 2011b]. The motivation for this work was to explore the possibility to use the eCST representation to compute metrics which reflect the complexity of ontological descriptions. In order to obtain the eCST representation of ontology, the SSQSA front-end has to be extended to support a language for the description of ontological systems. The aim of this paper is to explain how the SSQSA front-end is extended to support OWL2 language in functional-style syntax.

The rest of the paper is structured as follows. The next section presents the related work. Section 3 covers the integration of OWL2 into the SSQSA framework. In the next section are discussed the benefits of the eCST representation of ontology. The last section concludes the paper and gives directions for future work.

This work was partially supported by the Serbian Ministry of Education, Science and Technological Development through project *Intelligent Techniques and Their Integration into Wide-Spectrum Decision Support*, no. OI174023. The authors also would like to thank Rok Žontar for fruitful discussions on ontology metrics.
Author's address: M. Savić, Z. Budimac, G. Rakić, M. Ivanović, Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia, email: {svc, zjb, goca, mira}@dmi.uns.ac.rs; M. Heričko, Institute of Informatics, Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova ulica 17, 2000 Maribor, Slovenia, email: marjan.hericko@uni-mb.si.

In: Z. Budimac (ed.): Proceedings of the 2nd Workshop of Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA), Novi Sad, Serbia, 15.-17.9.2013, published at http://ceur-ws.org

2. RELATED WORK

2.1 Ontology metrics

In recent years, various metrics for measuring the complexity of ontological descriptions were proposed. Inspired by Chidamber and Kemerer [1994] metrics suite, Yao et al. [2005] proposed three cohesion metrics which are defined on a graph that represent subsumtion dependencies between ontological concepts. Orme at al. [2006] introduced three coupling metrics which are defined on the graph representation of ontology. Tartir et al. [2005] introduced OntoQA metric suite that contains 12 structural metrics also defined on ontological graph. Zhang et al. [2010] also proposed several new graph-based structural metrics for ontology evaluation. Their metrics suite, among others, contains metrics adopted from the Chidamber-Kemerer suite (NOC, DIT, CBO). Žontar and Heričko [2012] analyzed software metrics from the Lorenz-Kidd, Chidamber-Kemerer and Abreu metric suites in order to determine which of them can be adopted for ontologies. The results of their study show that graph-based software metrics can be adopted for ontology evaluation.

2.2 SSQSA Framework

The SSQSA framework consists of two parts, SSQSA front-end also known as eCST Generator, and the set of SSQSA back-ends, individual tools that operate on the eCST representation of source code. The main characteristic of eCST representation is that it contains so called universal nodes, language-independent markers that denote the meaning of concrete language constructs. The architecture of SSQSA is presented in Figure I. Also, it is shown how the architecture is planned to be extended with a new back-end in order support the analysis and evaluation of ontological systems.

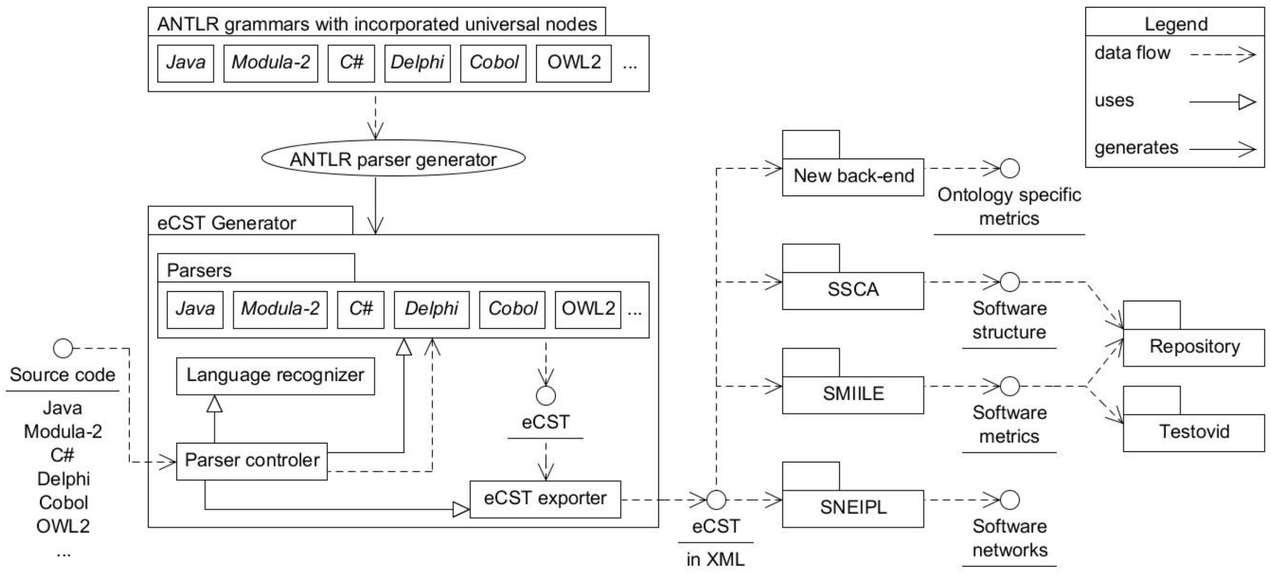


Figure I. Extended SSQSA architecture

SSQSA originated from a language-independent software metrics tool SMIILE [Rakić and Budimac 2011a]. SMIILE uses the eCST representation to calculate metrics reflecting internal complexity of software entities such as LOC and Cyclomatic complexity [McCabe 1976]. It is also integrated with Testovid, a semi-automated assessment system for students' programs, in order to provide metric-based qualification of programming assignments [Pribela et al. 2012]. SSCA was the first SSQSA back-end which extended the applicability of the eCST representation [Gerlec et al. 2012]. This tool tracks and analyzes changes in the hierarchical structure of software entities and stores its results in a repository that also contain metric values obtained using SMIILE. The last realized SSQSA back-end is SNEIPL [Savić et al. 2012]. This tool extracts dependency networks formed by software entities that can be used to

`

analyze the design complexity of software systems under the framework of complex network theory. Obtained networks can be also viewed as fact-bases required for reverse engineering activities and used to calculate metrics related to software design.

Currently SSQSA supports six general-purpose, imperative programming languages: Java, C#, Delphi, Modula-2, Pascal and Cobol. Therefore, this work is the first attempt to extend the SSQSA front end to produce the eCST representation of a declarative, domain-specific language.

3. INTEGRATION OF OWL2 LANGUAGE INTO THE SSQSA FRONT-END

eCST Generator uses parsers generated by the ANTLR [Parr and Quong 1995] parser generator to produce the eCST representation of source code that is provided as input. The advantage of using ANTLR to describe languages supported by SSQSA is the ANTLR grammar notation itself. This notation enables modification of syntax trees through tree rewrite rules that are attached to grammar productions. Therefore, in order to integrate OWL2 into the SSQSA front-end the following steps have to be made:

1. Realization of ANTLR grammar which describes OWL2 FSS,
2. Identification of OWL2 language constructs that corresponds to existing eCST universal nodes,
3. Incorporation of eCST universal nodes into tree-rewrite rules of the grammar in order to obtain eCST representation of parsed text.

3.1 Step 1 – ANTLR grammar for OWL2 FSS

The formal specification of OWL2 FSS in Extended Backus-Naur form (EBNF) can be found in the official W3C OWL2 language specification [Motik et al. 2012]. The ANTLR grammar notation closely follows EBNF, thus the grammar in [Motik et al. 2012] can be easily adopted for ANTLR. At this stage of the integration, the realized grammar is tested using ten ontologies from TONES[2] repository which are previously converted into OWL2 FSS using Protégé[3]. The results are summarized in Table I. It can be seen that the parser generated from the grammar successfully parsed more than 1.4 millions of lines of real-world ontological axioms in less than three minutes.

Table I. Results of testing of ANTLR grammar for OWL2 FSS using ontologies from TONES repository.

ONTOLOGY NAME	**LOC**	**Parse time [s]**
CTON (Cell Type Ontology)	144252	23
FMA (Foundational Model of Anatomy)	316101	41
Gene Ontology Edit	233608	22
Human Disease	476111	59
Teleost Taxonomy	182656	17
GEO Skills	20506	2
Matr Mineral	46	0.04
OBO Relation Ontology	25412	2
SC Ontology	23707	2
Software Ontology	5931	0.5

3.2 Step 2 – Universal nodes

OWL2 FSS language contains four types of tokens: keywords, separators, identifiers and constants. For each of mentioned lexical categories there are already introduced eCST universal nodes. Ontological axioms are marked with STMT universal node which is used to mark individual statements in imperative

[2] http://owl.cs.manchester.ac.uk/repository/
[3] http://protege.stanford.edu/

programming languages. Elements of an axiom are also marked with existing universal nodes (TYPE, ARGUMENT_LIST, and ARGUMENT). The PACKAGE_DECL universal node denotes that entities declared in an eCST sub-tree rooted at this node are mutually visible. Therefore, PACKAGE_DECL corresponds to the declaration of ontology. Declarations of ontological entities (concepts, roles and individuals) are marked with ATTRIBUTE_DECL universal node which is used to denote declarations of global variables in imperative programming languages. Ontological expressions that can be nested (class and data range expressions) are marked with the EXPR universal node.

OWL2 is a declarative, domain-specific language. Before the integration of OWL2, SSQSA supported several programming languages none of them being declarative or domain-specific. OWL2 axioms represent explicitly stated relations among ontological entities. Therefore, we introduced three new universal nodes that denote different categories of explicitly stated relations in general:

1. BINARY_RELATION (BR) marks binary relations
2. SYMMETRIC_RELATION (SR) marks symmetric n-ary relations
3. PARTIALLY_KNOWN_BINARY_RELATION (PKBR) marks binary relations in which one of the arguments is not known at the moment.

With BINARY_RELATION are marked all OWL2 relations that denote subsumptions and assertions. The SYMMETRIC_RELATION universal node is associated with relations indicating the equivalent and disjoints classes, same and different individuals, and equivalent and disjoint object properties. The PARTIALLY_KNOWN_BINARY_RELATION universal node marks object property domain and object property range relations. The newly introduced universal nodes are currently used only in the eCST representation of ontological descriptions. However, they can be used to mark explicitly stated binary and symmetric relations in other descriptive languages as well. Explicitly stated relations among entities in already supported imperative programming languages are marked with specific, more concrete universal nodes, such as EXTENDS and IMPLEMENTS. Those universal nodes can be viewed as sub-concepts of the BINARY_RELATION universal node.

3.3 Step 3 – Tree-rewrite rules

Once the correspondence between constructs of a concrete language and eCST universal nodes is identified, it is pretty straightforward to incorporate universal nodes into tree rewrite rules of the grammar. For example, it has been identified that ontology declarations correspond to the PACKAGE_DECL universal node. Therefore, PACKAGE_DECL universal node will be incorporated in the tree rewrite rule of the production that describe ontology declaration as the following excerpt from the OWL2 FSS grammar shows:

```
ontology :  'Ontology'  '(' (ontologyIRI versionIRI?)? importo* annotation* axiom* ')'
            -> ^(PACKAGE_DECL
                    ^(KEYWORD 'Ontology')
                    ^(SEPARATOR '(')
                    (ontologyIRI versionIRI?)?
                    importo* annotation* axiom*
                    ^(SEPARATOR ')')
                );
```

Besides the PACKAGE_DECL universal node, two other universal nodes are also incorporated in the rule: KEYWORD and SEPARATOR to mark keywords and separators in ontology declaration, respectively.

Figure II shows how a simple ontology named “PL” looks in the eCST representation. The complete description of the ontology in the functional-style syntax is as follows:

```
Ontology (:PL
     SubClassOf(:C :CPP)
)
```

The `SubClassOf` axiom states that each program written in the programming language C is at the same time valid C++ program.

`

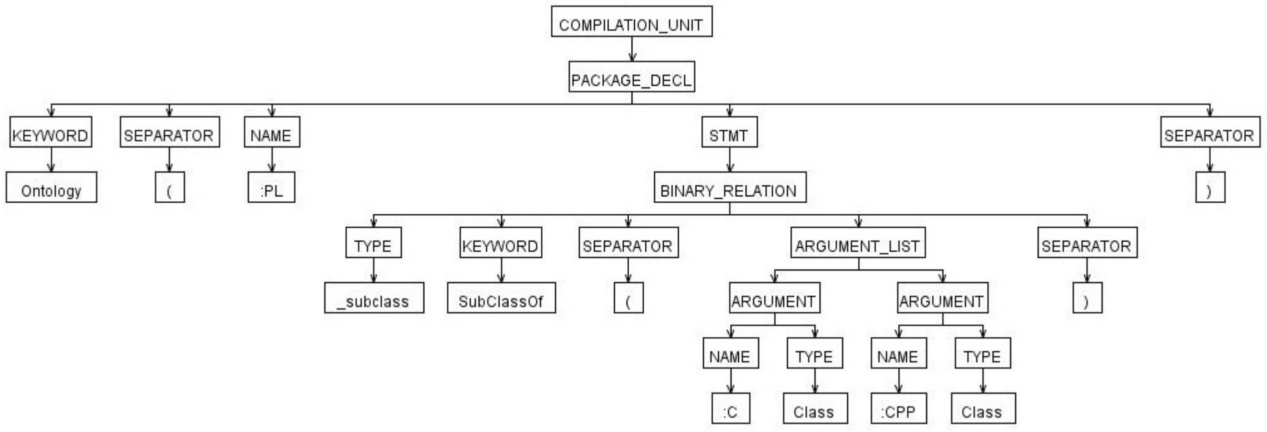


Figure II. eCST representation of a simple ontology.

4. BENEFITS OF OWL2 INTEGRATION INTO SSQSA

Metrics that reflect complexity of a description written in a programming or formal language can be classified as follows:

1. Metrics of internal complexity reflect lexical and syntactical complexity of the description or some of its parts. Lexical complexity measures are derived from the lexical elements of a language and reflect the complexity that is related to the volume of the description. Representative metrics which belong to this category are LOC family of metrics and Halstead [1977] complexity measures. Syntactical complexity is related to the compositional (structural) complexity of concrete language constructs. Cyclomatic complexity is an example of widely used measure of syntactical complexity.
2. Metrics of design complexity reflect the complexity of dependency structures among identifiers introduced in the description. Those metrics quantify inheritance, coupling and cohesion relationships among entities represented by the identifiers. Representative examples are CBO, NOC, DIT and LCOM metrics from the Chidamber-Kemerer metrics suite.
3. Hybrid metrics combine metrics of internal and design complexity. Examples are WMC and RFC from the Chidamber-Kemerer metrics suite, and the Henry-Kafura complexity [Henry and Kafura 1981].

As it can be seen from the review of related works on ontology metrics, the complexity of an ontological description is viewed as some measure of complexity of underlying graph representation. In other words, already introduced ontology metrics belong to the category of design complexity metrics. The integration of OWL2 into the SSQSA front-end provides the eCST representation of ontology. This representation can be used to define (or adopt) and compute metrics of internal complexity, which is not possible in the graph-based representation of ontology. For example, Halstead complexity metrics adopted for ontologies can be calculated in the same way as for software systems: by counting eCST universal nodes representing lexical categories. Similarly, the statement and expression level universal nodes can be used to derive syntactical complexity measures. Currently, it is possible to use the SMIILE back-end to obtain LOC and Halstead metrics for ontological descriptions. SMIILE also calculates cyclomatic complexity (CC) for software systems, but this metric cannot be adopted for ontology evaluation, since there are no OWL2 language elements that correspond to branch and loop statements. However, the predicate counting procedure used for the computation of the CC metric in SMIILE can be adopted to derive the complexity of nested OWL2 class and data range expressions (by counting EXPR universal nodes in eCST).

The ATTRIBUTE_DECL universal node can be used to recognize the declarations of ontologies, declarations of ontological concepts, roles and named individuals in represented ontological description. Relations among those entities can be identified by the analysis of eCST sub-trees rooted at BR, SR and

PKBR universal nodes (see Section 3.2). This means that the graph representation of ontology can be extracted from the eCST representation of ontology. Therefore, an ontology metrics tool that is based on the eCST representation also can be able to compute metrics of design complexity. Finally, metrics of internal and metrics of design complexity can be combined to obtain hybrid complexity metrics. The extraction of the graph representation of ontology is fundamentally different problem that the extraction of software networks due to the structural difference between ontological and software entities. The hierarchy tree representation of an ontological description can be obtained using the SNEIPL back-end, but SNEIPL cannot be used to identify horizontal dependencies (dependencies between entities of the same type) among ontological concepts and individuals. Those ontological entities are structurally atomic, i.e. they are not composed out of other ontological entities. To the contrary, software entities (classes, functions, etc.) are not structurally atomic: the definition of a software entity A associates the name of A with a body that contains the structure of A. Horizontal dependencies between software entity A and other entities are contained in the body of A, while horizontal dependencies between ontological entities are independent of ontological declarations. Since the SNEIPL back-end cannot be used to obtain the graph representation of ontology, a new SSQSA back-end that computes graph-based ontological metrics will be developed (see Figure I). This back-end will reuse and adopt modules from SMIILE to compute metrics of internal complexity, as well as modules from SNEIPL to form the hierarchy tree representation which is the first step in the extraction of ontological graph (identification of ontological entities and vertical dependencies).

The SSCA back-end constructs and compares hierarchy trees of two consequent versions of a software system in order to determine changes in vertical dependencies (dependencies among entities at different levels of abstraction). The hierarchy tree representation of an ontological description can be obtained from the eCST representation in the same way as for software systems: it is entirely determined by the hierarchical structure of eCST universal nodes in concrete eCSTs. This means that this back-end can be applied for ontologies in order to identify which concepts and named individuals are added or removed in the next version of ontology, and to what extent. Finally, with the design and development of new SSQSA back-ends it will be investigated whether they can be applied to analyze both software and ontological systems.

5. CONCLUSION AND FUTURE WORK

In this paper we described how the SSQSA front-end is extended to support OWL2 language in functional-style syntax. It is also shown that the eCST representation of ontologies can be used to compute metrics that reflect both internal and design complexity of ontological descriptions. Therefore, our future work will include the development of a SSQSA back-end, as shown in Figure I, that uses the eCST representation of ontology to compute metrics reflecting different aspects of complexity of ontological descriptions. In our future work we will also investigate whether recently introduced metrics of cognitive complexity of programs written in object-oriented languages [Misra et al. 2012] and metrics of complexity of web services descriptions [Basci and Misra 2012] can be adopted and used for ontology evaluation.

REFERENCES

Dilek Basci and Sanjay Misra. 2012. Metric suite for maintainability of eXtensible Markup Language web services. *IET Softw.* 5, 3, 320-341.

Zoran Budimac, Gordana Rakić, and Miloš Savić. 2012. SSQSA architecture. In *Proceedings of the Fifth Balkan Conference in Informatics* (BCI '12). ACM Conf. Proc. 1479, 287-290.

Shyam R. Chidamber and Chris F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng.* 20, 6, 476-493.

Črt Gerlec, Gordana Rakić, Zoran Budimac, and Marjan Heričko. 2012. A programming language independent framework for metrics-based software evolution and analysis. *Computer Science and Information Systems* 9, 3, 1155-1186.

Thomas R. Gruber. 1993. A translation approach to portable ontology specifications. *Knowl. Acquis.* 5, 2, 199-220.

Maurice H. Halstead. 1977. *Elements of software science.* Elsevier North-Holland, Amsterdam.

Sallie M. Henry and Dennis G. Kafura. 1981. Software structure metrics based on information flow. *IEEE Computer Society Trans. Software Engineering* 7, 5, 510-518.

`

Thomas J. McCabe. A Complexity Measure. 1976. *IEEE Trans. Software Eng.* 2(4): 308-320.

Sanjay Misra, Murat Koyuncu, Marco Crasso, Cristian Mateos and Alejandro Zunino. 2012. A Suite of Cognitive Complexity Metrics. In *Computational Science and Its Application ICCSA* 2012. Lecture Notes in Computer Science, Vol. 7336 Springer Berlin Heidelberg, 234-237.

Boris Motik, Peter F. Patel-Schneider and Bijan Parsia. 2012. OWL2 web ontology language structural specification and functional-style syntax (second edition). Retrieved July, 2013 from http://www.w3.org/TR/owl2-syntax/

Anthony M. Orme, Haining Yao, and Letha H. Etzkorn. 2006. Coupling Metrics for Ontology-Based Systems. *IEEE Softw.* 23, 2, 102-108.

Terence J. Parr and Russell W. Quong. 1995. ANTLR: a predicated-LL(k) parser generator. *Softw. Pract. Exper.* 25, 7, 789-810.

Ivan Pribela, Gordana Rakić, and Zoran Budimac. 2012. First Experiences in Using Software Metrics in Automated Assesssment. In *Proc. of the 15th International Multiconference on Information Society* (IS), *Collaboration, Software and Services in Information Society* (CSS), Vol. A, 250-253.

Gordana Rakić and Zoran Budimac. SMIILE Prototype. 2011a. In *Proc. Of International Conference of Numerical Analysis and Applied Mathematics* ICNAAM2011*, Symposium on Computer Languages, Implementations and Tools* (SCLIT), AIP Conf. Proc. 1389, 853-856.

Gordana Rakić and Zoran Budimac. Introducing Enriched Concrete Syntax Trees. 2011b. In *Proc. of the 14th International Multiconference on Information Society (IS), Collaboration, Software and Services in Information Society (CSS)*, Vol. A, 231-234.

Miloš Savić, Gordana Rakić, Zoran Budimac, and Mirjana Ivanović. 2012. Extractor of software networks from enriched concrete syntax trees. In *Proc. Of International Conference of Numerical Analysis and Applied Mathematics* ICNAAM2012, *Symposium on Computer Languages, Implementations and Tools* (SCLIT), AIP Conf. Proc. 1479, 486-489.

Samir Tartir, Budak I. Arpinar, Michael Moore, Amith P. Sheth, and Boanerges Aleman-Meza. 2005. OntoQA: Metric-based ontology quality analysis. In *Proceedings of IEEE Workshop on Knowledge Acquisition from Distributed, Autonomous, Semantically Heterogeneous Data and Knowledge Sources*.

Hongyu Zhang, Yuan-Fang Li, and Hee Beng Kuan Tan. 2010. Measuring design complexity of semantic web ontologies. *J. Syst. Softw.* 83, 5, 803-814.

Rok Žontar and Marjan Heričko. 2012. Adoption of object-oriented software metrics for ontology evaluation. In *Proceedings of the Fifth Balkan Conference in Informatics* (BCI '12). ACM Conf. Proc. 1479, 298-301.

Mobile Device and Technology Characteristics' Impact on Mobile Application Testing

TINA SCHWEIGHOFER AND MARJAN HERIČKO, University of Maribor

Mobile technologies have a significant impact on processes in ICT, including software development. Within mobile technologies a new type of software has emerged: mobile applications. Nowadays, the concept of mobile applications is widely known and the development of mobile applications is more and more widespread. One of the most important parts of mobile application development is mobile applications testing. The testing process has always been very important and crucial in the software development cycle, which is why testing constitutes an important aspect of software development. An appropriate testing procedure significantly increases the quality level of the developed product. With mobile application development testing, new challenges associated with mobile technologies and device characteristics, have arisen. Some examples of these challenges are: connectivity, convenience, touch screen technology, context awareness, supported devices, etc. It is important that we adequately address these challenges and perform an appropriate mobile application testing process, resulting in a high quality product without critical defects that could cause quality issues or the unwanted waste of human or financial resources. In this paper, we will present a mobile application testing process. We will indicate the important parts and especially emphasize the challenges related to mobile devices and technology features and properties.

General Terms: Mobile applications testing

Additional Key Words and Phrases: testing, mobile applications, mobile technologies, quality

1. INTRODUCTION

Mobile devices and mobile applications play an important role in our everyday lives. Nowadays we are surrounded by mobile technology and cannot imagine running personal or business errands without them. This has been confirmed by numerous pieces of research. According to Gartner, the worldwide sale of mobile phones in the third quarter of 2012 reached almost 428 million units. Within this number, smartphone sales represent almost 40 percent of total mobile phone sales [Gartner 2012]. A similar thing is happening in the area of mobile subscriptions. At the end of 2012, there were approximately 6.8 billion mobile subscribers in the world, which is equal to 96 percent of the world population. Currently, global mobile-cellular penetration rates are 96 percent. In Europe the number is higher, at 126 percent [ITU 2013].

Closely related to mobile devices are mobile applications. By the end of 2012, there were approximately 1.1 million mobile applications users. According to forecasts, the number will grow rapidly – by nearly 30 percent per annum - to reach 4.4 billion by the end of 2017 [Whitfield 2013a]. Applications generated $12 billion in revenue in 2012 and a total of 46 billion applications were downloaded [Portio Research 2012]. This number is also expected to grow: in 2013 smartphone and tablet users will download a further 82 billion applications [Whitfield 2013b]. Mobile applications are currently represented in almost every possible personal or business domain. Although games still constitute the largest category in most of the major application stores [Whitfield 2013b], mobile applications can be seen in just about every industry. Some examples include: retail, media, travel, education, healthcare, finance, social, business applications, collaboration and more [uTest 2012]. Some of these applications within a specific

domain use more or less sensitive user data. Users frequently allow access to personal data in the context of mobile devices and also enter a lot of personal information. In this context, the issue of users' trust takes on an important role. It becomes important to provide quality mobile applications that are reliable and flawless [Hu and Neamtiu 2011]. Applications that are reliable and work flawlessly within expected functionalities can gain a user's trust and, more importantly, keep it. Users also often have high

Author's address: T. Schweighofer, Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova 17, 2000 Maribor, Slovenia; email: tina.schweighofer@uni-mb.si; M. Heričko, Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova 17, 2000 Maribor, Slovenia; email: marjan.hericko@uni-mb.si.

In: Z. Budimac (ed.): Proceedings of the 2nd Workshop of Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA), Novi Sad, Serbia, 15.-17.9.2013, published at http://ceur-ws.org

expectations about the quality of mobile applications. Applications that crash and lose users' personal data are not allowed [Bo et al. 2007]. One of most important mechanisms for providing reliable, flawless and quality mobile applications is an appropriate testing procedure. Testing during mobile application development is slightly different from testing procedures in traditional software and the process itself is also suited to the area of mobile applications and mobile technologies.

In this paper we will present a testing procedure for testing mobile applications. We will identify and describe specific characteristics for mobile devices, mobile applications and mobile technologies as a whole, which have a significant impact on the testing procedure. First, in Section 2, we will present the fundamentals of software testing and reveal some of the major differences between testing traditional and mobile software. We will also provide an introduction to mobile application testing. In Section 3, we will present some of the specific characteristics of mobile technologies that have an impact on testing and challenges in testing mobile applications. Everything will be cemented with a practical approach for mobile application testing procedures and gained experiences. In the Discussion, we will present the findings and results of our work.

2. FUNDAMENTALS OF MOBILE APPLICATION TESTING

Mobile application development has specific characteristics that need to be addressed through the entire product's life cycle. According to a recent study [Wasserman 2010], there are important software engineering research issues linked to mobile application development. Some of these issues include: potential interaction with other applications, handling available sensors, the development of native or hybrid mobile applications, different families of hardware and software mobile platforms, problems of security, an adjusted user interface and the problem of power consumption.

Testing process plays an important role in the life cycle of a software product, whether in mobile or traditional desktop application. Therefore, it is crucial to address abovementioned issues in related mobile testing procedures.

A lot of research has dealt with the fundamentals of software testing, therefore there are many available definitions of testing. To summarize one of the definitions: testing is an activity performed for the purpose of evaluating product quality, and for improving the product by identifying potential defects and problems. Software testing is composed of the dynamic verification of the program behavior on a finite set of test cases against the expected program behavior [Bourque and Dupuis 2004].

Testing is not just an activity that starts after the coding phase is finished and is used to detect failures. Software testing is a procedure that should be active through the entire product life cycle, from the development and maintenance process to actual product construction. Also, the planning phase for testing should occur early in the product requirements process and test plans must be systematically and continuously developed, as the development of a product proceeds. Currently it is considered that the right strategy for quality is one of prevention. It is much better to avoid problems than to correct them. Therefore, testing must be viewed as a procedure for checking if prevention was successful and for identifying faults in cases where prevention was not effective [Bourque and Dupuis 2004].

An important aspect that makes mobile testing different is the complexity of testing, a point made by the authors of the aforementioned study [Wasserman 2010]. A challenge that they mention is the diversity of different available mobile devices, for example Android devices and others related to testing native mobile applications. There are also many other challenges related to mobile application testing. We will describe these challenges in detail in the subsection below.

2.1 Mobile application as testing object

If we want to properly understand the concept of mobile application testing, it is important that we understand what a mobile application is. We are all familiar with mobile applications, but what does the definition say? A mobile application is a type of software application designed to run on smart phones, tablets and other mobile devices and/or for taking in input information. Similarly, mobile applications in the context of mobile computing is an application that runs on an electronic device that may move [Kirubakaran and Karthikeyani 2013].

The testing of mobile applications is an important and also very difficult task, according to various authors [Bo et al. 2007; She et al. 2009; Kirubakaran and Karthikeyani 2013; Franke and Weise 2011]. They all believe that testing mobile applications is a non-trivial process that takes a lot of time, effort and other resources. We have had the same experience with projects where we developed mobile applications for Android, iOS and BlackBerry. The experience is described in detail below in Section 3. As previously mentioned, as mobile applications become more and more complex and ubiquitous, users have higher and higher expectations with regard to mobile application quality. Users want an application that does not fail, lose data or harm the device's operability, as well as applications that are secure, reliable and easy to use. If we conduct the testing procedure properly, possible defects embedded in the application can be detected and removed and this can lead to greater confidence in an application [Bo et al. 2007; She et al. 2009].

The challenges encountered during mobile application testing were mostly related to the different characteristics of mobile devices or mobile technologies, which has a direct influence on mobile applications and the conducted testing procedure. In the existing literature we found many different described characteristics. As noted by [Kirubakaran and Karthikeyani 2013; Franke and Weise 2011] these characteristics are: connectivity, convenience, user interface, supported devices, touch screens, new programming languages, resource constraints, context awareness and data persistence. The mentioned characteristics are presented in Figure 1.

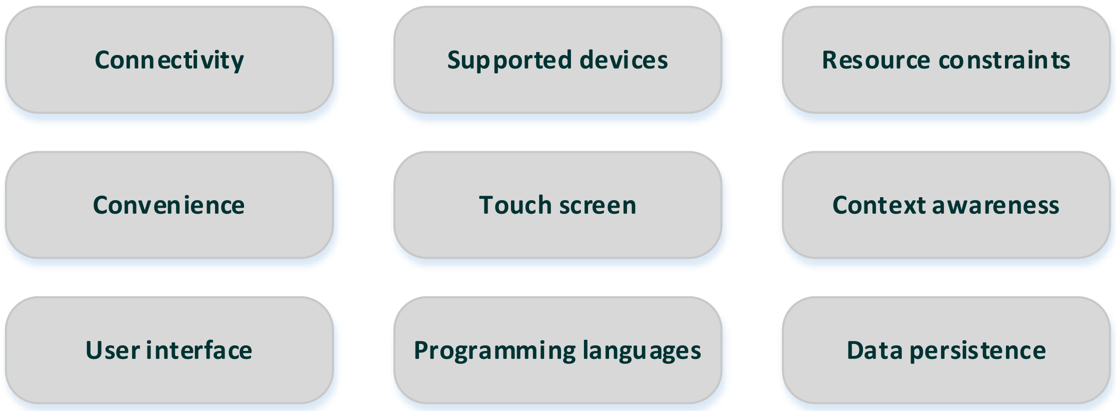


Fig. 1. Characteristics of mobile devices and technologies with their impact on the testing procedure

3. CHALLENGES IN MOBILE APPLICATION TESTING

As previously mentioned, during mobile application testing we came across different challenges. Different authors have already investigated some of the challenges that have a significant influence on the testing procedure. We came across the same characteristics that consequently represent challenges in testing mobile applications. As mentioned, we developed mobile applications for the operating systems Android, iOS and BlackBerry in the context of a research and development project. Mobile applications are a part of the larger project, which also include a web application. Within the development process, we also perform mobile application testing. The process of application testing is a complex process, but for the needs of this article we will show a simplified version. The simplified testing process can be seen in Figure 2. The process starts with the release of a version of the mobile application for a specific platform for testing purposes. The Quality Assurance team receives aversion and starts the process of testing based on the recorded test scenarios. If they find an irregularity, an error or an unreliable function, they report the problem to the web-based bug tracking system. Bugs are seen by the development team and later fixed. We have to point out that within our project, we also performed different types of test cycles. The most common was the weekly testing procedure. There is also testing for the purpose of the application's release on the belonging market.

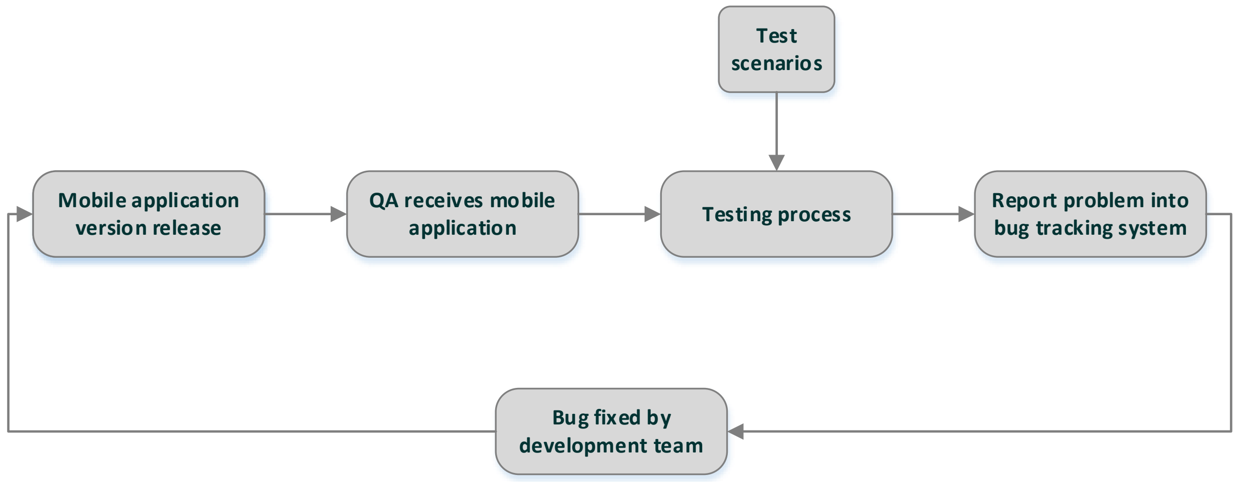

Test scenarios
Mobile application version release
QA receives mobile application
Testing process
Report problem into bug tracking system
Bug fixed by development team

Fig. 2. Mobile application testing procedure

The most important part of the testing process is the execution of test scenarios, where specific characteristics of mobile devices are revealed. In fact, they also play an important part in writing testing scenarios, where we have to shape each test scenario in a way that it will consider and verify a specific characteristic. When we started to write and later execute specific test scenarios, we reviewed existing literature from the area of mobile application testing. Specific characteristics identified in different works were taken into account within our own testing procedure. The nature of these characteristics, what existing literature says, and how we dealt with them is discussed below.

The first property we came across and has an impact on many different types of testing is *connectivity*. Mobile applications have to be designed with the awareness that they will be always online, because mobile devices are always logged on to a mobile network. Networks can vary according to speed, reliability and security. Especially slow and unreliable wireless networks are a common obstacle for mobile applications. The described property has to be considered in functional testing, where different networks and connectivity scenarios have to be performed, with an emphasis on popular networks. Connectivity also has an effect on performance, security and reliability testing [Kirubakaran and Karthikeyani 2013; uTest 2012]. In practice, we consider the characteristic *connectivity* in such a way that we test our applications in different networks. We also perform test scenarios to test different internet connections. We use different Wi-Fi networks and cellular networks by different operators and in different places, like buildings, city centers or in nature. For our application, connectivity is very important because functions in mobile applications are supplemented with web applications, so the application uses the function of synchronization very often.

Another important property according to other studies is the *user interface*, which is related to the characteristic of *convenience*. This property is important because user interfaces in development need to follow specific guidelines based on the different platforms for which they are being developed. Different platforms have their own rules and guidelines about how a specific user interface should look, so if a product in development is being developed for different platforms we have to strongly focus on a specific design. Regardless, different platforms still present a big challenge in terms of designing the best possible use of limited screen space, so that the design of the user interface takes greater importance in the development process. The user interface looks different based on the mobile device's screen resolution and its dimensions. Some implications on testing are seen in the area of different devices that need to be used for testing procedure. It is recommended to test the user interface on as many different mobile device as possible. This is because different devices behave differently with the same application code [Hu and

Neamtiu 2011; Kirubakaran and Karthikeyani 2013; Wasserman 2010]. Within the development of mobile applications in our project, the developers followed specific rules and good practices for designing platform specific applications. These guidelines were also reviewed in the testing phase. We also developed our own Style guide document, which ensured that regardless of the platform, the application would look similar and reflect the fact that all applications are part of the same product family. With regard to the testing process, we tested the appearance on different mobile phones, with different resolution and different physical dimensions. We considered the minimal and optimal screen size, which was set within the Software requirements specification document.

Nowadays many different mobile devices are available. What is important is that applications work flawlessly on as many devices as possible. *Supported devices* represent one of the most difficult aspects of the testing process. Devices from different vendors have different software and hardware components. In particular, there are hundreds of different mobile devices that run the operating system Android, whereas the mentioned operating system has countless different versions. Different versions of operating systems are also a great challenge to cover within the testing process [Kirubakaran and Karthikeyani 2013]. Usually it is impossible to test every available device, so we group mobile devices in different categories, as proposed in [Kirubakaran and Karthikeyani 2013]. The focus of this challenge is on Android mobile devices. We tested our mobile applications on mobile devices from different vendors, with different hardware components and different versions of operating systems. We developed three groups: small, optimized and high quality mobile devices. The first group included mobile devices with a small screen size and low resources, while the last group included mobile devices with a high screen resolution and a lot of resources. Test scenarios were carried out on a few representatives of each group. However, iOS devices were a different story as there is not such a large variety of different mobile devices. The same testing strategy was used for testing the *touch screens* of mobile devices and their properties, which also represent an important challenge in mobile application testing. Touch screens are the main tool for inputting user data into a mobile application. An important aspect is the system response time to a touch, which depends on device resource utilization, and easily may become slow in some circumstances, such as in the case of a busy processor, a lack of memory or other problem. Thus, it is important to test the touch screen's abilities under different circumstances [Kirubakaran and Karthikeyani 2013]. We tested touch screen capabilities under different circumstances, as proposed. We burdened the processor and available memory by running multiple applications simultaneously, for the purpose of testing the behavior of different touch screen on different devices.

As many authors agree, mobile devices are becoming more and more powerful, but their *resources*, like processor power, RAM, and resolution are still facing restrictions [Kirubakaran and Karthikeyani 2013; She et al. 2009; Franke and Weise 2011; Portio Research 2012]. This characteristic is closely linked to some of the previously mentioned characteristics, like *supported devices* and *touch screens*. As proposed in [Kirubakaran and Karthikeyani 2013] mobile device resources have to be continuously monitored, to see what a specific mobile device is capable of and to verify what actions are taken if a device runs out of resources. A very similar characteristic is *data persistence*, because mobile applications that run out of memory shut down running applications, so we have to make sure user data is stored and saved adequately [Franke and Weise 2011]. We also test these two characteristics within specified groups of mobile device testing. We try to overload a specific mobile device and test the behavior of a mobile application. We check if it stored data properly and of course where the breaking limit for the mobile application is.

A very important characteristic that has a significant impact on testing our mobile application is *context awareness*. A lot of mobile applications also rely on sensed data, provided by context providers that monitor the surroundings and connectivity of devices. All these provide an enormous amount of data, which vary depending on the user's actions and the environment. It is important to test the application under a different environment and under any contextual input, if it is going to work correctly [Kirubakaran and Karthikeyani 2013]. Our application uses data provided by GPS sensors and via Bluetooth from heart rate sensors. We have to ensure that the data is provided correctly regardless of the mobile device and its operating systems. Different operating systems support different Bluetooth devices so we have to ensure that we test all available and supported devices properly.

The characteristic that is more involved in the developing process, but still part of the testing process, is related to new *programming languages* that are used for mobile application development. These programming languages were developed to support mobility, managing resource consumption and handling new GUIs [Kirubakaran and Karthikeyani 2013]. It is important that code during the development process is tested properly, according to the features and characteristics of programming languages.

DISCUSSION

All characteristics were, as mentioned, appropriately represented in the process of writing test scenarios, and later considered in the testing of developed mobile applications. The majority of characteristics are important after the development phase is finished and it is time to test the developed software for defects and irregularities. Software testing is an activity aimed at evaluating the quality of a program and also for improving it by identifying defects and problems, as claimed in [Kirubakaran and Karthikeyani 2013]. As we emphasized, some important characteristics that have a significant impact on mobile application testing are: connectivity, convenience, user interface, supported devices, touch screens, new programming languages, resource constraints, context awareness and data persistence. All characteristics found by different authors were appropriately considered while testing our mobile application for the operating systems Android, iOS and BlackBerry, developed within a research and development project. In the future, we would like to spread those characteristics into automatic testing and test management tools for the mobile application domain.

Testing mobile devices is different from testing a mobile application. We think that mobile application testing cannot be properly conducted if the characteristics specific to mobile devices and mobile technologies are ignored. If we test mobile application regardless mentioned characteristics we will find obvious defects, but irregularities specific to mobile application will not be discovered. And we will not be able to offer a quality product to potential users. The mentioned characteristics and guidelines have a significant impact on testing and consequently on mobile application quality. And as we know, a quality product means satisfied and loyal users.

REFERENCES

BO, J., XIANG, L. AND XIAOPENG, G., 2007. MobileTest: A Tool Supporting Automatic Black Box Test for Software on Smart Mobile Devices. *Second International Workshop on Automation of Software Test (AST '07)*, pp.8–8.

BOURQUE, P. AND DUPUIS, R., 2004. Guide to the Software Engineering Body of Knowledge. *Guide to the Software Engineering Body of Knowledge, 2004. SWEBOK.*

FRANKE, D. AND WEISE, C., 2011. Providing a Software Quality Framework for Testing of Mobile Applications. *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pp.431–434.

GARTNER, 2012. Gartner Says Worldwide Sales of Mobile Phones Declined 3 Percent in Third Quarter of 2012; Smartphone Sales Increased 47 Percent.

HU, C. AND NEAMTIU, I., 2011. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*. New York, NY, USA: ACM, pp. 77–83.

ITU, 2013. The World in 2013 - ICT Facts and Figures.

KIRUBAKARAN, B. AND KARTHIKEYANI, V., 2013. Mobile application testing — Challenges and solution approach through automation. *2013 International Conference on Pattern Recognition, Informatics and Mobile Engineering*, pp.79–84.

PORTIO RESEARCH, 2012. Your Portio Research Mobile Factbook 2012.

SHE, S., SIVAPALAN, S. AND WARREN, I., 2009. Hermes: A Tool for Testing Mobile Device Applications. *Software Engineering Conference, 2009. ASWEC '09. Australian*, pp.121–130.

UTEST, 2012. The Essential Guide to Mobile App TestingNo Title.

WASSERMAN, A.I., 2010. Software engineering issues for mobile application development. *Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER '10*, p.397.

WHITFIELD, K., 2013a. Fast growth of apps user base in booming Asia Pacific market. *Portio Research.*

WHITFIELD, K., 2013b. What apps are people using? *Portio Research.*