Developer Modelling using Software Quality Metrics and Machine Learning

Franciele Beal, Patricia Rucker de Bassi and Emerson Cabrera Paraiso
*Graduate Program in Informatics, Pontifícia Universidade Católica do Paraná,
Rua Imaculada Conceição 1155, Curitiba, Brazil*

Keywords: User Modelling, Machine Learning, Quality Metrics, Supervised Learning.

Abstract: Software development has become an essential activity for organizations that increasingly rely on these to manage their business. However, poor software quality reduces customer satisfaction, while high-quality software can reduce repairs and rework by more than 50 percent. Software development is now seen as a collaborative and technology-dependent activity performed by a group of people. For all these reasons, choosing correctly software development members teams can be decisive. Considering this motivation, classifying participants in different profiles can be useful during project management team's formation and tasks distribution. This paper presents a developer modeling approach based on software quality metrics. Quality metrics are dynamically collected. Those metrics compose the developer model. A machine learning-based method is presented. Results show that it is possible to use quality metrics to model developers.

1 INTRODUCTION

Software development has become an essential activity as organizations increasingly rely on it to manage their business. The increasing use of systems in broader contexts of business makes its construction a complex activity. Several experts must work together to develop successful software, which depends on many factors, such as compliance on time, on estimated cost and the quality desired by the customer. According to a study conducted by the University of Cambridge (Cambridge University, 2013), source code defects correction annual cost is around US$312 billion. A low-quality code impacts on delivery schedules, repairs and rework that become project's major cost drivers. Moreover, low quality reduces customer satisfaction, can affect market share, and in some cases, can even lead to criminal charges. On the other hand, a high-quality software reduces repairs needs and rework, sometimes by more than 50%. Furthermore, costs linked to application's maintenance and support are also reduced. Consequently, having a high-quality software also improves testing and delivery schedules (Bettenburg and Hassan, 2013); (Bonsignour and Jones, 2011).

Some researchers such as Mohtashami et al., (2011) and Whitehead et al., (2010) show that software development is a collaborative activity dependent on technology and performed by a group of people. It usually involves people playing different roles, such as managers, software architects, developers, testers and requirements engineers. All these professionals work with different types of artefacts in different activities; most of them are activities that require teamwork and collaboration. These development environments follow computer supported collaborative work (CSCW) principles (Grudin, 1994) where participants construct artifacts such as source code, diagrams, requirements documents in a collaborative way. Finally, common sense believes that greater collaboration between participants increase chances of reaching final product faster and in better condition (Whitehead et al., 2010); (Magdaleno et al., 2015).

Successful organizations use to measure their main activities as part of their day-to-day tasks (McGarry et al., 2002). This measurement process has played an increasingly important role in Software Engineering. Software metrics allow measurement and evaluation, controlling the software product and processes improvement (Fenton and Neil, 2000) (Kitchenham, 2010). They are essential resources to improve quality and cost

Beal, F., Bassi, P. and Paraiso, E.
Developer Modelling using Software Quality Metrics and Machine Learning.
DOI: 10.5220/0006327104240432
In *Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS 2017) - Volume 1*, pages 424-432
ISBN: 978-989-758-247-9

control during software development (Wallace and Sheetz, 2014).

In general, collaborative software development processes rank participants in different skill's levels to facilitate team building and task assignments involving software source code programming. It is common the use of subjective criteria to generate this classification. However, to assign a class to a team member is important to guide the project management. For example, it is important to recognize developer' classes to form productive collaborative programming pairs in Extreme Programming (XP) agile methodology (Padberg and Muller, 2003). Pairs formed by different experience classes of developers can learn from each other through knowledge sharing (Aiken, 2004). Thus, it is important to know each developer's features to be able to form pairs where developers can effectively contribute with each other exchanging knowledge and experience.

Considering these evidences and the importance that developer's classification represents in software development process, we propose in this study a way to automatically class developers. We advocate that developers may be classified according to their performance and productivity. To do so, we are using a set of software quality metrics to compose the developer's model. In real time, we collect those metrics and use them to model each developer. In addition, once we have the actual code quality overview, we can indicate to each developer, code improvements tips in real time as well.

In this paper, we present a machine learning based approach to model developers. The approach was evaluated and the results are presented.

This article is organized as follows: in section 2 related works are presented, section 3 presents the background on user modeling and applications. The features we are using to model a developer are described in section 4. Section 5 presents a machine learning-based method for modeling developers. We detail experiments to test our proposed method for developer modeling in section 7 and finally we present conclusions and future work.

2 RELATED WORKS

User Modeling (Kobsa, 2001) is essential for customized services. A user model (or profile) structures a set of features and preferences that is used as a basis to adapt or recommend some type of information to a user or a group of user (Rich, 1983). It is applied in different domains where there is a great information volume and someone want to infer knowledge about users and provide personalized results to meet their needs. Amazon (www.amazon.com) and Netflix (www.netflix.com) are well-known e-commerce sites examples that apply user profiles to make items recommendations (as books and movies). This kind of sites apply for each user histories browsing, purchased items lists, viewed items, interested subjects and issues, among others, to recommend relevant items to each of them.

User modelling are also applied to customize searches and filters (Gemechu et al., 2010); (Veningston and Simon, 2011); (Leung et al., 2013) and (Jingqiu et al., 2007), personalized news recommendation (Kyo-Joong et al., 2014) and (Won-Jo et al., 2014), adaptive hypermedia systems (Kavcic, 2000) and learning systems (Virvou et al., 2012). We can also cite the use of user modeling in the social networks (Piao and Breslin, 2016), personal assistants (Guha et al., 2015), adaptive advertising (Qaffas and Cristea, 2015), personalized cloud computing recommendation (Zou et al., 2014), among others.

There are a few studies applying user modeling in software engineering. Galete and colleagues (2013) developed a system to cluster developers based on data gathered by Hackystat (Johnson et al., 2004). The authors used a Kohonen's Map to cluster them. The main problem on that approach is that it is not possible to classify developers since the content on each cluster may change from one execution to another.

The lack of research on user modeling in the context of software development lead us to propose a system to model developers. The next section presents the background on user modeling useful to understand the approach we propose in this paper.

3 USER MODELING

One of the main goals of user modeling is the customization and adaptation of systems to the user's specific needs. Another purpose, more related to our work, is modeling specific kinds of users, including their skills and declarative knowledge.

A user model is a structured representation of features and data about a user. Among the different design patterns for user models, we highlight two: static modeling and dynamic modeling (Johnson and Taatgen, 2005).

Static user models are the most basic kind of user models. Once the main data is gathered they are not normally changed. Shifts in users' preferences are not registered and learning algorithms are not used to alter the model.

Dynamic user models allow a more up to date representation of users. Changes in their interests, their learning progress or interactions with the system are noticed and influence the user models. The models can thus be updated and take the current needs and goals of the users into account. In these kind of models, data are automatically, implicitly, and non-invasively collected, through sensors that monitor user's interaction and behavior. We are using the dynamic model to model developers.

The dynamic user model construction process consists basically of data that will compose the model and a few steps presented in the next paragraphs (Barth, 2010).

In the first step, named Model Composition, one identifies the features (characteristics, preferences, interests, knowledge, and so on) that will compose the model. These features are domain dependent.

The next step, the Model Acquisition, performs data acquisition using sensors that monitor the environment where users are emerged. This process is fully automatic. In the context of code development, Hackystat (Johnson et al., 2004) provides sensors to collect data generated by developers while they code. Hackystat sensors capture real-time developer's programming events and store them in a database without developer's intervention. The implicit data acquisition mechanism makes it suitable for user's dynamic modeling.

After data acquisition comes the Model Induction step. In general, supervised machine learning algorithms are used to induct a model from historical data. Algorithms like Decision Trees and SVM (Nguyen, 2009), Naïve Bayes (Wen et al., 2008) or Artificial Neural Networks (Chen and Norcio, 1991) are used. The main challenge in supervised learning is to find available labeled data to be used in the training step of the induction process. In our context, this inducted model will be used to evaluate the data collected in real time from a given user in order to class the user according to a specific type (more details in section 4).

Once a machine learning model is inducted, starts the Applying Model step where the model is used to infer the actual user type.

Finally, the Maintaining Model is performed to keep the user model updated. In a dynamic user model, the system makes it automatically, monitoring and observing user's actions. Maintenance involves the re-train of a machine learning based algorithm.

The section 5 presents a method for developers modelling based on these steps. The next section presents the features we are using to model a developer and consequently to compose the user model.

4 THE DEVELOPER MODEL

The previous section presented the user model composition. The first step is to identify the set of features that will compose the user model. The set of features is gathered during the user interaction with the environment where he takes part of. In the context of software development and, in special, source code implementation, the environment is an IDE (Integrated Development Environment). Moreover, we intend to model developers with respect to the quality of their final work: source code. Thus, we are proposing a set of features related to software quality metrics. The idea is simple: to collect and to evaluate, in real time, a set of 20 object-oriented (OO) metrics. In addition to these metrics, we are using data related to good programming practices and developer's programming errors. We assume that this set of features can be used to model a developer and also to estimate the quality of his work.

Source code quality can be measured by software metrics regard to complexity, testability, reusability and maintainability. These quality software metrics seek to manage and reduce structural complexity, improve maintainability and source code development, and consequently the software itself (Yu and Zhou, 2010). Studies such as (Filó et al., 2015); (Silva et al., 2012); (Li, 2008); (Oliveira et al. 2008); (Olague et al., 2006),; (Anderson, 2004); (Horstmann, 2005), and (Oliveira et al., 2008) report that source code metrics may indicate situations of code quality increasing or decreasing regard to maintainability, testability, reusability and complexity.

The set of 20 metrics, shown in Table 1, were grouped in four different categories: complexity metrics, inheritance metrics, size metrics and coupling metrics. The first category groups complexity metrics, considering that more complex classes and methods may turn the source code more difficult to understand, maintain and test ((McCabe, 1976), (Chidamber and Kemerer, 1994), (Henderson-Sellers, 1996) and (Martin, 2002)). The following metrics: McCabe's cyclomatic complexity metric, weighted methods per class metric, lack of cohesion in method metric, and nested block depth, can indicate code complexity situation ((Anderson, 2004), (Olague et al., 2006); (Oliveira et al., 2008) and (Horstmann, 2005)).

The second category groups the following metrics: Depth of inheritance tree, number of children, number of overridden methods and specialization index metrics. They form the inheritance category, since they can indicate abstraction and software maintenance problems ((Chidamber and Kemerer, 1994); (Schroeder, 1999) and (Henderson-Sellers, 1996)).

The size metrics like method lines of code, number of attributes per class, number of static attributes, number of static methods, number of parameters, number of interfaces and number of packages can indicate understanding, maintenance and reuse code problems (Henderson-Sellers, 1996); (Harrison et al., 1997).

Finally, strong coupling in software turn more difficult to maintain, modify and reuse code. Coupling can compromise code abstractness and stability. Afferent and efferent coupling, instability metric, abstractness and normalize distance from main sequence are metrics that can indicate code coupling (Martin, 2002).

Table 1: Software Quality Metrics Excerpt.

Metric influence on code quality	**Metric**
Complexity Metrics	McCabe's Cyclomatic complexity metric - MCC
More complexes code/class/method more difficult to understand, maintain and test.	Weighted Methods per Class metric - WMC
	Lack of Cohesion in Method metric-LCOM*
	Nested Block Depth – NBD
Inheritance Metrics	Depth of Inheritance Tree - DIT
Can indicate problems of abstraction and software maintenance.	Number of Children – NOC
	Number of Overridden Methods – NORM
	Specialization Index - SIX
Size Metrics	Method Lines of Code -MLOC
Can indicate problems of understanding, maintenance and reuse.	Number of Attributes per Class - NOA
	Number of Static Attributes - NSF
	Number of Static Methods - NSM
	Number of Parameters - NOP
	Number of Interfaces - NOI
	Number of Packages – NOP
Coupling Metrics	Afferent Coupling - Ca
Strong coupling in software is more difficult to maintain, modified, reused.	Efferent Coupling – Ce
	Instability metric - I, named here as RMI
Can compromise code abstractness and stability.	Abstractness - A, named here as RMI
	Normalize Distance from Main Sequence – D

Each metric in table 1 has its own range that can be used to evaluate the real state of the source code. For instance, the metric McCabe's Cyclomatic (MCC) measures the complexity of the code. It has the following range:

low: $1 \leq \text{MCC} \leq 10$
moderate: $10 < \text{MCC} \leq 20$
high: $20 < \text{MCC} \leq 50$
very high: $\text{MCC} > 50$

In this case, the ideal situation should place this metric in the *low* level (Anderson, 2004).

In addition to the metrics, data related to programming practices and developer's programming errors were used as features. They are: *Debug, Breakpoint, Refactoring, Code Error* and *Number of Errors*. The *Debug* feature is used to indicate whether the developer has used the debugger or not. *Breakpoint* indicates whether the breakpoint feature was used. *Refactoring* indicates whether refactoring has occurred, that is, whether a method, attribute, or class has been renamed, removed, or moved. The *Code Error* feature refers to code errors committed by developers at coding time. *Number of errors* indicates the total number of errors committed for a code error.

The next section presents the method for developer modeling based on these features.

5 DEVELOPER MODEL CONSTRUCTION

In this section, we present a machine learning based method for modeling developers. The method consists of different steps respecting the dynamic modeling process presented in section 3. Figure 1 shows all four steps.

It is important to note that the method is based on the principle that developers can be modeled according to data gathered online during coding. Once the data is available the system can class a developer in one of *n* different classes. In our context, a class represents the programming skill level of a developer. Even if the number of classes may be irrelevant for our approach, we fixed the number of programming skills in 3 levels (classes): *basic*, *intermediate*, and *proficient*. Thus, at the end of Step 3 (model application) a developer is classified as a basic developer, intermediate developer, or proficient developer. The level is a combination of all features (metrics). Thus, the quality (*Q*) of the source code produced by a developer classified as *basic* is worse than the one classified as *proficient*: $Q_{basic} < Q_{intermediate} < Q_{proficient}$.

We assume that historical data is available for classifying developers according to this classification.

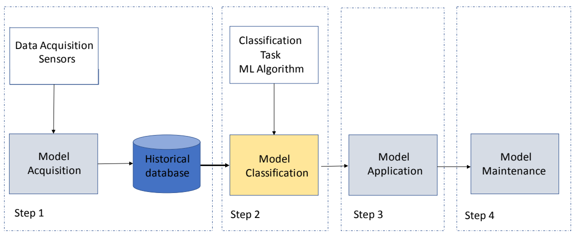

Figure 1: The method overview.

The first step, Model Acquisition, refers to collecting data to compose the developer's model. The data is gathered during coding by sensors. Sensors should be linked to the IDE developers are using in a non-invasive way (as shown in Figure 2). Two type of data are collected: code quality metrics and programming practice. A new instance is created in the database every time something changes in the developer model: a metric value or a programming practice event.

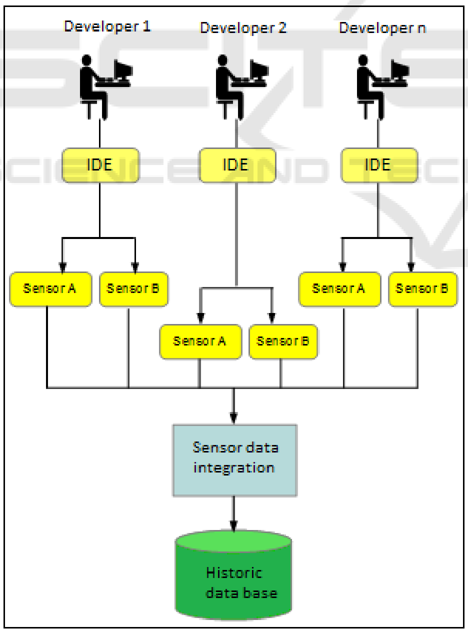

Figure 2: Data collection process.

The resultant database is used to train a supervised machine learning algorithm. A supervised algorithm needs labeled data. This means, for each instance, to label it as one of the defined classes. In our case: basic, intermediate, or proficient. To the best of our knowledge there is no database available in the literature labeling developers according to their programming skills. To be able to evaluate our approach we developed our own database. Details are given in section 6.

The second step, named Model Classification, trains (inducts) a learning algorithm to build a classifier model. The model is used to classify developers into different classes (levels). The train process uses the labeled database produced in the previous step. A classifier must be chosen. A validation schema, such as cross validation should be set. We present in the next section an experiment evaluating different classifiers with different configurations. In our case, the decision tree algorithm C4.5 (Quinlan, 1993) achieved the best results.

The third step, Model Application, should be also integrated as a plugin to the IDE used by developers. The plugin allows developers to be classified in real time. As sensors collect new developer's features (which are added to the historical database), the data is used to evaluate the developer. Figure 3 shows schematically how it is done.

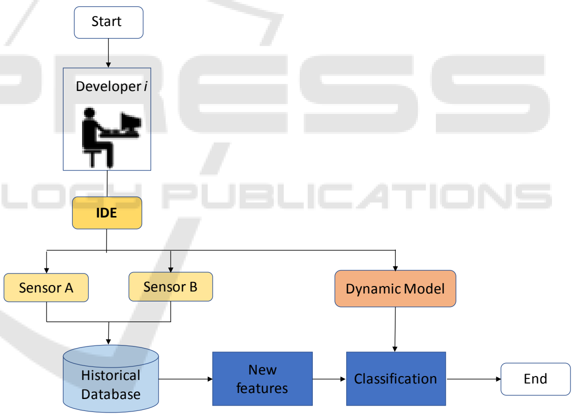

Figure 3: Dynamic model application.

At the end of the process a developer *i* receives a label (*basic*, *intermediate*, or *proficient*). The classification of a specific developer can change over time since a new classification is done every time the code is changed. The class assigned to a developer is a consequence of his contribution to the source code.

The last step, Model Maintenance, should be executed to re-train the model that classifies the developers. It is the opportunity to improve the accuracy of the model using new data generated continuously by developers. It is recommended to execute this step every time a new project begins.

6 DEVELOPER MODEL DATABASE CONSTRUCTION

In this section, we detail experiments to test our proposed method for developer modeling. A Java prototype was implemented using Hackystat (Johnson et al., 2004) sensors (to gather programming practice data) and the plugin Metrics for Eclipse (www.metrics2.sourceforge.net) (to gather software quality metrics data).

Different classifiers were evaluated: k-Nearest Neighbors algorithm (k-NN), C4.5, Naive Bayes, Multilayer Perceptron (MLP) and Support Vector Machines (SVM). The WEKA version of each algorithm was used in the prototype (Hall et al., 2009). All classifiers were trained and evaluated using cross-validation.

We have mentioned in the foregoing that, to the best of our knowledge, there is no database available in the literature labeling developers according to their programming skills. This motivated us to develop our own database. The database must have data enough to be used in the model induction. Unfortunately, it is not simple to find several voluntary professional developers to participate in the process of gathering quality metrics data. Thus, we invited 57 undergraduate computer science students to help in this process. All of them were enrolled in the Bachelor of Computer Science at the Pontifícia Universidade Católica do Paraná - Brazil (PUCPR). This undergraduate course has six consecutive programming courses (one course per semester). All six courses have Java as the main programming language. From the total of 57 students, 19 students were finishing the second semester, 19 students were finishing the fourth semester and 19 students were finishing the sixth semester. They were labeled respectively as basic, intermediate, and proficient. The basic level represents less experienced students with up to 1 year of Java language programming. Intermediate level concentrates students with 1 to 2 years of language programming, and proficient level concentrates the most experienced students, with Java language knowledge around 3 to 4 years.

Each participant received a two-page document specifying a system to be coded. The system was very simple to code: an application to compute the body mass index (BMI). A section of two hours was set to code the system. Each developer had available an instance of Eclipse (www.eclipse.org) configured with a plug-in that gathered all metrics and data generated during code (set of features presented in section 4). The number of instances generated for each developer varies according to their programming style, coding speed, and so on. Table 2 shows the number of instances for each group of developers.

Table 2: Instances distribution per programming skill.

Programming Skill	**Instances**
Basic	665
Intermediate	847
Proficient	679
Total	**2,191**

The Database stores data instances, where each instance corresponds to a developer profile that has been collected over time. Data instances are composed by the following fields: WMC – Weighted Methods per Class metric, LCOM* - Lack of Cohesion in Method metric, NBD – Nested Block Depth metric, DIT – Depth of Inheritance Tree metric , NOC – Number of Children metric, NORM – Number of Overridden Methods metric, SIX – Specialization Index metric, MLOC – Method Lines of Code metric , NAC – Number of Attribute per Class metric, NSF – Number of Static Attribute metric, NSM – Number of Static Methods metric, PAR – Number of Parameter metric, NOI – Number of Interfaces metric, NOP – Number of Package metric, I – Instability metric, A – Abstractness metric, Dn – Normalize Distance from Main Sequence metric, Debug, Breakpoint, Refactoring, ErrorCode, ErrorQty. Quality metrics data are collected using Metrics and programming practice events are collected using Hackystat. Thus, each instance is composed by 22 characteristics plus developer's level (*basic, intermediate or proficient*).

The database is available at: https://www.ppgia.pucpr.br/~paraiso/DesenvolvimentoColaborativoDeSoftware/completo_perfil_desenvolvedor.arff.

7 EXPERIMENTS AND RESULTS

First, we evaluated the selected classifiers. Classifiers were evaluated using 10-fold Cross Validation. All parameters were set to their default values. Table 3 shows the accuracy for each classifier. After statistical analysis, the C4.5 algorithm proved be the best for the task of classifying developers according to their skills. Table 4 shows the overall f-measure results for every programming skill.

Table 3: Classifiers accuracy.

Classifiers	Accuracy
C4.5	90.96%
MLP	86.44%
3-NN	85.08%
SVM	79.37%
Naive Bayes	65.91%

Table 4: F-measure: class X classifier.

Classifier	F-Measure			
	basic	*intermediate*	*proficient*	Weighted avg.
C4.5	**0.899**	**0.934**	**0.890**	**0.910**
MLP	0.857	0.879	0.854	0.865
3-NN	0.838	0.872	0.837	0.851
SVM	0.806	0.785	0.791	0.793
Naive Bayes	0.726	0.559	0.666	0.634

Results are promising and give us good indications that developer modeling can be used in real cases. These results need to be confirmed when new databases become available.

The results among classes (programming level) are uniform. The C4.5 algorithm achieved an accuracy of 90.96%. The lowest classification error was registered for the basic level. The confusion matrix shows that are some confusing involving intermediate developers and basic developers. This is probably due to the fact that some students enrolled in the fourth semester were also enrolled in the second semester. The confusion matrix also showed that some intermediate students (developers) were confused with proficient students. This is due to the fact that some students in the fourth semester had professional experience (as developers) in the industry.

We inspected the rules generated in the decision tree (C4.5). The quality metrics is more important than the programming practice data. However, in misclassified instances those features (programing practice) seems to be relevant. We are working in this analysis and the results will be subject of future publication.

8 CONCLUSIONS AND FUTURE WORKS

High-quality software reduces repairs needs and rework. Successful organizations use to measure their main activities as part of their day-to-day tasks (McGarry et al., 2002). Software metrics allow measurement and evaluation, controlling the software product and processes improvement (Fenton and Neil, 2000); (Kitchenham, 2010). In this study, we proposed a machine learning-based approach to model developers using quality software metrics. Developers are modelled on the fly. A set of quality metrics was selected to compose the developer model. The model enables a less subjective classification of developers. It allows a better distribution of tasks among developers. It also allows a precise analysis of the code generated by each developer. The method is non-invasive.

Experiments were carried out and showed that our approach is promising. Considering our results it is possible to build a developer dynamic model based on source code data produced by him. Through code quality metrics and programming practices events research results indicate the possibility of developers skills level different classification.

In the future, we intend to extend the existing sensors and use new sensors to collect other developer's features. It is our intend to apply this experiment with developers from industry. We also plan to use the actual value of quality metrics to give tips on how to evolve the code to improve each metric.

Resultant rules generated by the decision trees are also part of our future works to better understand the role of each feature in classification process

Finally, we are working in a new method to be used with legacy code. In this scenario, a distributed version-control platform is used to collect source-code.

ACKNOWLEDGEMENTS

The authors would like to thank Fundação Araucária/Brazil and CAPES/Brazil (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) for their support in this research (grant number 075/2013).

REFERENCES

Aiken, J., 2004. Technical and human perspectives on pair programming. In *SIGSOFT Software Engineering Notes*.

Anderson, J., 2004. Using software tools and metrics to produce better quality test software. In *AUTOTESTCON 2004. Proceedings, IEEE.*

Barth, F. J., 2010. Modelando o perfil do usuário para a construção de sistemas de recomendação: um estudo teórico e estado da arte. *In Revista de Sistemas de Informação.*

Bettenburg, N., Hassan, A. E., 2013. Studying the impact of social interactions on software quality, Empirical Software Engineering.

Bonsignour, O., Jones, C., 2011. The Economics of Software Quality, Addison Wesley.

Cambridge University, Cambridge University study, http://insight.jbs.cam.ac.uk/2013/financial-content-cambridge-university-study-states-software-bugs-cost-economy-312-billion-per-year/, 2013. Accessed: Sep 10th 2016.

Chen, Q., Norcio, A.F., 1991. A neural network approach for user modeling. In *1991 IEEE International Conference on Systems, Man, and Cybernetics, Decision Aiding for Complex Systems.*

Chidamber, R. S,. Kemerer, C. F., 1991. Towards a metrics suite for object-oriented design. In *the OOPSLA 91 Conference.*

De Silva, D., Kodagoda, N., Pereira, H., 2012. Applicability of three complexity metrics. In *2012 International Conference on Advances in ICT for Emerging Regions (ICTer).*

Fenton, N E., Neil, M., 2000. *Software metrics: Roadmap*. In *Conference on The Future of Software Engineering, ICSE '00, ACM*, New York, NY, USA.

Filo, T,. Bigonha, M., Ferreira, K., 2015. A catalogue of thresholds for object-oriented software metrics. In *1st International Conference on Advances and Trends in Software Engineering, IARIA.*

Galete, L., Ramos, M. P., Nievola, J. C., Paraiso, E. C., 2013. Dynamically modeling users with MODUS-SD and Kohonen's map. In *2013 IEEE 17th International Conference on Computer Supported Cooperative Work in Design (CSCWD).*

Gemechu, F., Zhang Yu Liu Ting, 2010. A Framework for Personalized Information Retrieval Model. In *2010 2nd International Conference on Computer and Network Technology (ICCNT).*

Grudin, J., 1994. Computer-supported cooperative work: history and focus. In *IEEE Computer.*

Guha, R,. Gupta, V., Raghunathan, V., Srikant, R., 2015. User Modeling for a Personal Assistant. In *8thACM International Conference on Web Search and Data Mining (WSDM '15).*

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I. H., 2009. The WEKA Data Mining Software: An Update. SIGKDD Explorations.

Harrison, R., Counsell, S., Nithi, R, 1997. An overview of object-oriented design metrics. In *8th IEEE International Workshop on Software Technology and Engineering Practice.*

Henderson-Sellers, B., 1996. Object-Oriented Metrics: Measures of Complexity. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Horstmann, C., 2005. Big java: Programming and practice, John Willey & Sons.

Jingqiu Xu, Zhengyu Zhu, Xiang Ren, Yunyan Tian, Luo Ying, 2007. Personalized Web Search Using User Profile. In *International Conference on Computational Intelligence and Security.*

Johnson, A., Taatgen, N., 2005. User Modeling, Handbook of human factors in Web design, Lawrence Erlbaum Associates.

Johnson, P. M., Kou, H., Agustin, J. M., Zhang, Q., Kagawa, A. and Yamashita, T., 2004. Practical Automated Process and Product Metric Collection and Analysis in a Classroom Setting: Lessons Learned from Hackystat-UH. In *International Symposium on Empirical Software Engineering (ISESE'04).*

Kavcic, A., 2000. The role of user models in adaptive hypermedia systems. In *Electrotechnical Conference, 2000. MELECON 2000. 10th Mediterranean.*

Kitchenham, B., 2010. What's up with software metrics? A preliminary mapping study. In *Journal of Systems and Software, SI: Top Scholars.*

Kobsa, A., 2001. Generic User Modeling Systems. The Adaptive Web.

Kyo-Joong Oh, Won-Jo Lee, Chae-Gyun Lim, Ho-Jin Choi, 2014. Personalized news recommendation using classified keywords to capture user preference. In *2014 16th International Conference on Advanced Communication Technology (ICACT).*

Leung, K.W.-T., Dik Lun Lee, Wang-Chien Lee, 2013. PMSE: A Personalized Mobile Search Engine. In *IEEE Transactions on Knowledge and Data Engineering.*

Li, H., 2008. A novel coupling metric for object-oriented software systems. In *IEEE International Symposium on Knowledge Acquisition and Modeling Workshop, KAM Workshop 2008.*

Magdaleno, A M., Barros, M O., Werner, C. M. L., Araujo, R M., Batista, C. F. A., 2015. Collaboration in process optimization software composition. In *The Journal of Systems and Software.*

Martin, R. C., 2002. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall.

McCabe, T. J. A., 1976. Complexity measure. In *IEEE Transactions on software Engineering.*

McGarry, J., Card, D., Jones, C., Layman, B., Clark, E., Dean, J., Hall, F., 2002. Practical Software Measurement: Objective Information for Decision Makers. Addison Wesley.

Mohtashami, M., Ku, C. S., Marlowe, T. J., 2011. Metrics are needed for collaborative software development. In The *Journal of Systemics, Cybernetics and Informatics.*

Nguyen, L. A, 2009. Proposal of Discovering User Interest by Support Vector Machine and Decision Tree on Document Classification. In *International Conference on Computational Science and Engineering. CSE '09.*

Olague, H. M., Etzkorn, L. H., Cox, G. W., 2006. An entropy-based approach to assessing object-oriented software maintainability and degradation-a method and case study. In *Software Engineering Research and Practice.*

Oliveira, M. F., Redin, R. M., Carro, L., da Cunha Lamb, L., Wagner, F. R., 2008. Software quality metrics and their impact on embedded software, In *5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software.*

Padberg, F., Muller, M. M., 2003. Analyzing the cost and benefit of pair programming. In *9th International Software Metrics Symposium.*

Piao, G., Breslin, J. G., 2016. Analyzing Aggregated Semantics-enabled User Modeling on Google+ and Twitter for Personalized Link Recommendations. In *2016 Conference on User Modeling Adaptation and Personalization (UMAP '16). ACM.*

Qaffas, A. A., Cristea, A. I., 2015. An Adaptive E-Advertising user model: The AEADS approach. In *2015 12th International Joint Conference on e-Business and Telecommunications (ICETE).*

Quinlan, J. R., 1993. C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers.

Rich, E., 1983. Users are individuals: - individualizing user models.

Schroeder, M., 1999. A practical guide to object-oriented metrics, *IT professional 1*.

Veningston, K., Simon, M., Collaborative filtering for sharing the concept based user profiles. In *2011 3rd International Conference on Electronics Computer Technology (ICECT).*

Virvou, M., Troussas, C., Alepis, E., 2012. Machine learning for user modeling in a multilingual learning system. In *2012 International Conference on Information Society (i-Society).*

Wallace, L. G., Sheetz, S. D., 2014. The adoption of software measures: A technology acceptance model (tam) perspective. In *Information & Management.*

Wen, H., Fang, L., Guan, L., 2008. Modelling an individual's Web search interests by utilizing navigational data. In *2008 IEEE 10th Workshop on Multimedia Signal Processing.*

Whitehead, J., Mistrik, I., Grundy, J., 2010. Collaborative software engineering: concepts and techniques, In: Mistrík I, Grundy J, van der Hoek A, J Whitehead (eds.) Collaborative Software Engineering, Springer.

Won-Jo Lee, Kyo-Joong Oh, Chae-Gyun Lim, Ho-Jin Choi, 2014. User profile extraction from Twitter for personalized news recommendation. In *2014 16th International Conference on Advanced Communication Technology (ICACT).*

Yu, S., Zhou, S., 2010. A survey on metric of software complexity. In 2010 *The 2nd IEEE International Conference on Information Management and Engineering (ICIME).*

Zou, G., Gan, Y., Zheng, J., Zhang, B., 2014. Service composition and user modeling for personalized recommendation in cloud computing. In *2014 International Conference on Computing, Communication and Networking Technologies (ICCCNT).*