Predicting Software Metrics at Design Time

Wolfgang Holz[1], Rahul Premraj[1], Thomas Zimmermann[2], and Andreas Zeller[1]

[1] Saarland University, Germany, {holz, premraj, zeller}@st.cs.uni-sb.de
[2] University of Calgary, Canada, tz@acm

Abstract. How do problem domains impact software features? We mine software code bases to relate problem domains (characterized by imports) to code features such as complexity, size, or quality. The resulting predictors take the specific imports of a component and predict its size, complexity, and quality metrics. In an experiment involving 89 plug-ins of the ECLIPSE project, we found good prediction accuracy for most metrics. Since the predictors rely only on import relationships, and since these are available at design time, our approach allows for early estimation of crucial software metrics.

1 Introduction

Estimating the cost (or size) for a software project is still a huge challenge for project managers—in particular because the estimation typically is done at a stage where only few features of the final product are known. To date, several models have been proposed to improve estimation accuracy [1], but none have performed consistently well. Moreover, although a lot of emphasis is laid upon early estimation of development costs, the parameters used by many models are not known until much later in the development cycle [2]—that is, at a stage when prediction is both trivial and worthless.

In this work, we show how to reliably predict code metrics that serve as inputs (including software size) to prediction models very early on in the project by *learning from existing code*. We leverage the *problem domain of the software* to predict these metrics. The problem domain manifests itself in *import relationships*—that is, how individual components rely on each other's services. In earlier work, it has been shown that a problem domain, as characterized by imports, impacts the likelihood of software defects [3] or vulnerabilities [4].

Our approach is sketched in Figure 1. We train a learner from pairs of imports and code metrics, as found in existing software. The resulting predictor takes the set of imports for a component (as available in the design phase) and predicts metrics for the component. Managers can then use the predicted metrics as a basis to make other decisions, such as: *What will this product cost to develop? How many people should I allocate to the project? Will this product have several defects to be fixed?*

This paper makes the following contributions:

1. We present a novel software size estimation method during the design phase of development.

2. Using the ECLIPSE code base, we show how imports can be used to predict *software size*, given as source lines of code (SLOC) [5].
3. Using the ECLIPSE code base, we show how to predict *software complexity*, as defined by the widely used object-oriented *ckjm* software metrics [6].

We expect that advance reliable knowledge of such product-specific metrics can be a boon to solving several management issues that constantly loom over all types of development projects at an early stage.

This paper is organized as follows. In Section 2, we discuss features and shortcomings of contemporary cost estimation models. The data used for our experimentation is elaborated upon in Section 3. Thereafter, we present our experimental setup in Section 4, which is followed by results and discussions in Section 5. Threats to validity are addressed in Section 6 and lastly, we conclude our work in Section 7.

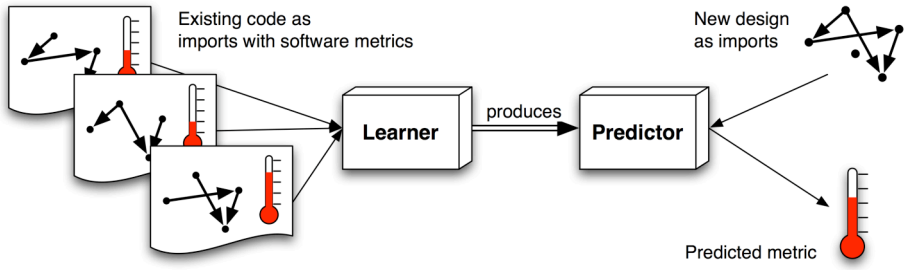

Fig. 1. Approach overview. By learning from the relationship between imports and metrics in existing code, we can predict metrics based on imports alone.

2 Background

As discussed above, cost estimation is vital to a successful outcome of a software project. But most contemporary estimation models depend upon characteristics of the software that are typically unknown at start. For example, many models take into account the relationship between software size and cost. Examples include algorithmic models such as COCOMO [7] and Putnam [8], regression models [9] and analogy-based models [10–13]. To use these models, first an estimate of the size of the project is needed. Again, size is unknown at start of the project and can only be estimated based on other characteristics of the software. Hence, basing cost estimates on an estimate of size adds to uncertainty of the estimates and fate of the project. This challenges the value of such models.

We propose a novel approach that, in contrast to others, focusses on estimating the size of a component with as little knowledge as its design. This places managers at a unique position from where they can choose between several alternatives to optimize not only size, but also other metrics of the software that serve as its quality indicators. We present these metrics in more detail in the following section.

3 Data Collection

We used 89 core plug-ins from the ECLIPSE project as data source for our study. Core plug-ins are those that are installed by default in ECLIPSE. We used both source code and binaries to extract the data necessary to build prediction models. In this section, we describe the metrics extracted and the methods employed for their extraction. The metrics or features can be grouped into two categories; first, *input features*, i.e., the features that are already known to us, and second, *output features*, which we wish to predict.

3.1 Input Features

As mentioned above, we hypothesize that the domain of the software determines many of its metrics, for instance, defects—a quality indicator. Similar to Schröter et al. [3], we assume that the *import directives* in source code indicate the domain of the software.

Naturally, our first task is to establish the domains of the 89 ECLIPSE plug-ins, i.e., extract all import directives from the relevant code. At first, this task seems trivial because one can quickly glance through JAVA source code to find the import directives at the top of the file.

However, this task becomes very complex when one encounters a situation as illustrated in Figure 2. Here, the import directive in Label 1 contains reference to package `import java.sql.*` instead of classes. Later, in Label 2, objects of classes `Connection` and `Statement` belonging to the `java.sql` package have been instantiated.

It is crucial that such references to packages are resolved to class levels; else we run the risk of leading statistical learning models astray. To accomplish this, we used the Eclipse *ASTParser* [14] that transforms JAVA code into a tree form, where the code is represented as AST nodes (subclasses of *ASTNode*). Each element in JAVA code has an associated, specialised AST node that stores relevant information items. For example, a node *SimpleType* contains the *name, return type, parameters*, and like. Further information to examine the program structure more deeply is allowed by *bindings*, a provision in *ASTParser*. It is these bindings that resolve import packages into the relevant classes. Figure 2 demonstrates this where the two classes referred to in Label 2 get resolved by the *ASTParser* as `java.sql.Connection` (Label 3) and `java.sql.Statement` (Label 4) respectively.

Using the above method, we extracted 14,185 unique and resolved import statements from the 89 ECLIPSE plug-ins used in this study.

3.2 Output Features

As mentioned earlier, the knowledge of as many product-specific metrics early in the project's life cycle has several advantages. We demonstrate our model's capacity to predict such metrics on a set of commonly known and used in the software development community.

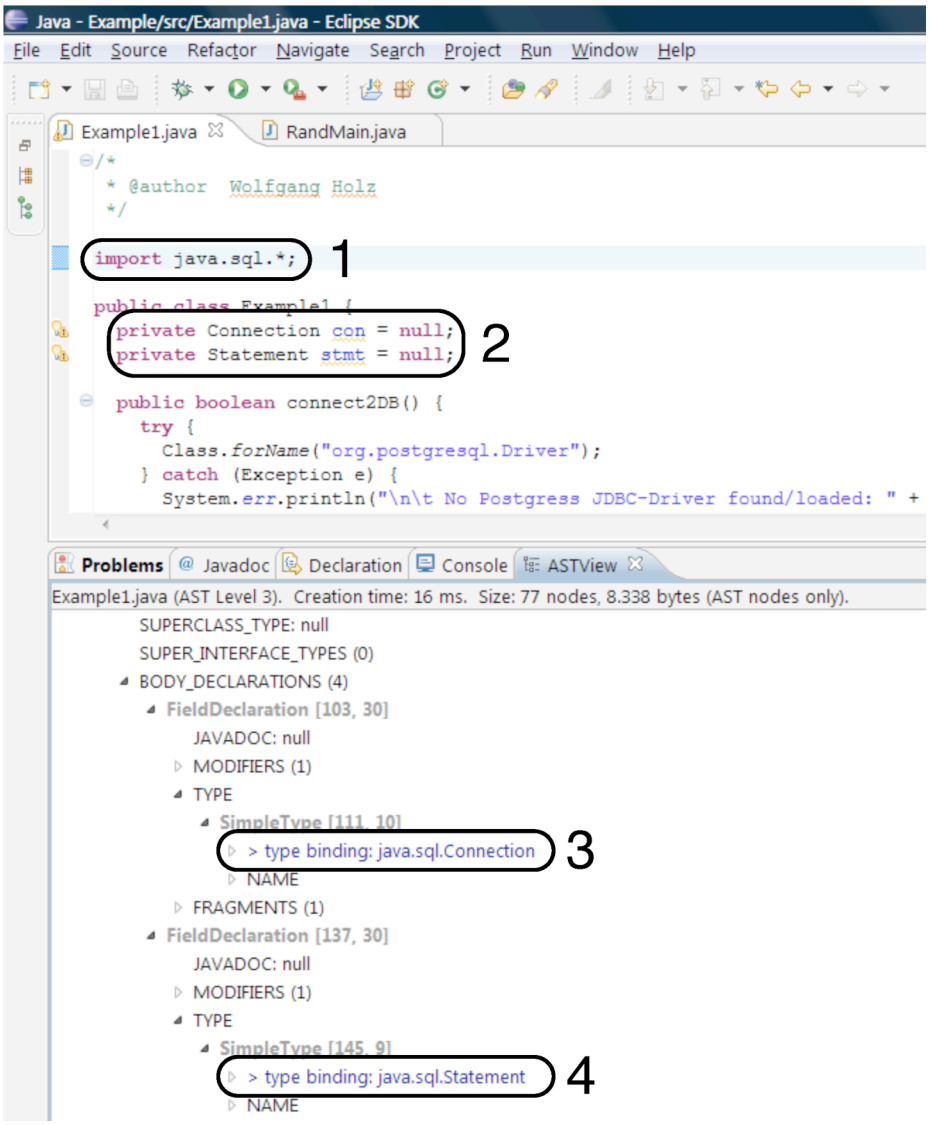

Fig. 2. An illustration of the use of the *ASTParser* to resolve import directives.

Source Lines of Code (SLOC). The count of lines of code is the simplest measure of the system size. Early estimate of SLOC or a similar size measure can substantially influence management and execution of the project: development costs and duration of the project can be estimated, system requirements can be inferred, required team size can be appropriated, and like.

Many definitions for counting SLOC have been proposed. We implemented a tool to count SLOC abiding the guidelines laid by Wheeler [5]. As Wheeler recommends, we count the *physical lines of code*, which is defined as follows:

> A physical SLOC is a line ending in a new line or end-of-file marker, and which contains at least one non-whitespace non-comment character.

Object-Oriented (OO) Metrics. Our second output feature is a set of OO metrics, referred to as *ckjm* metrics defined by Chidamber and Kemerer [6]. The *ckjm* tool computes six different metrics, summarised in Table 1. These metrics have previously been used to predict fault-proneness of classes [15], changes in short-cycled development projects using agile processes [16], system size [17, 18], and as software quality indicators [19–21].

Table 1. List of *ckjm* Metrics

Abbreviation	Metric
CA	Afferent Couplings
CBO	Coupling between Class Objects
CBOJDK*	Java specific CBO
DIT	Depth of Inheritance Tree
NOC	Number of Children
NPM	Number of Public Methods
LCOM	Lack of Cohesion in Methods
RFC	Response for a Class
WMC	Weighted Methods per Class

* In this metric, Java JDK classes (java.*, javax.*, and others) are included. We created a new metric because the use of JDK classes does not count toward a class's coupling because the classes are relatively stable in comparison to the rest of the project.

While the *ckjm* metrics have been shown to be useful predictors of a variety of software characteristics, a downside of their usage is that substantial parts of the code have to be written to reliably compute them. At this juncture, when code is reasonably mature, the value of such predictions is diminished, i.e., the new knowledge arrives too late in the product's life cycle. Our research alleviates this problem by predicting the *ckjm* metrics for classes at a very early stage of the life cycle. Endowed with predicted values of the *ckjm* metrics, project managers can make further predictions of software characteristics based on these values.

In Table 2, we present some summary statistics of the output features. The values suggest that most metrics are highly skewed. DIT is somewhat flat and most classes have no children (NOC), similar to the finding by Chidamber and Kemerer [6]. In fact, almost 84% of the classes had no children. Particularly noticeable is the fact that many metrics have extreme outliers, for example maximum value of LCOM is 329,563.

Table 2. Summary Statistics of Output Features

Metric	Min	Max	Median	Mean	Std. Dev
CA	0	588	2	5.40	5.23
CBO	0	212	9	13.86	16.15
CBOJDK	2	223	15	20.40	18.56
DIT	1	8	1	1.67	1.05
NOC	0	82	0	0.47	2.03
NPM	0	834	4	7.13	13.26
LCOM	0	329,563	9	164.10	3200.28
RFC	0	848	24	39.46	48.96
SLOC	3	7645	72	146.70	273.64
WMC	0	835	7	12.02	18.06

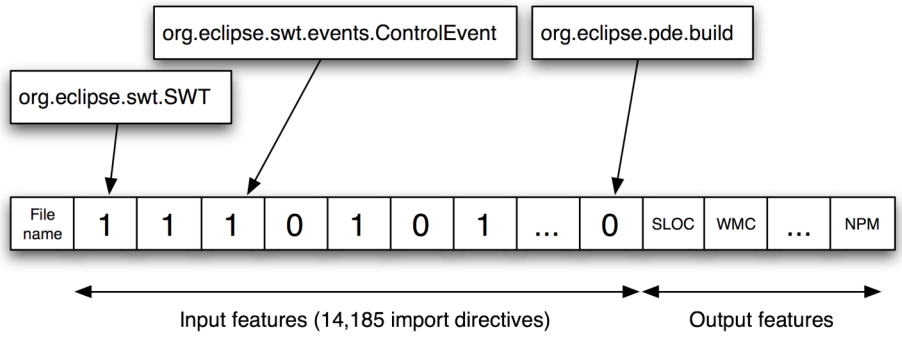

Fig. 3. Data format for experimentation

3.3 Data Format

After the data has been extracted, we have to shape it as feature vectors to be fed into statistical learning models. Each file is represented as a single row. The input features, that is, the imported classes are represented as dummies. This is illustrated in Figure 3 where each of the 14,185 import directives is represented as one column. To indicate that a file imports a class, the value of the respective cell is set to 1, while otherwise it is set to zero. The eleven output features (SLOC and *ckjm* metrics) are represented as columns too alongside the input features. As a result, we have a large matrix with 11,958 rows (files) and 14,196 columns (filename + input features + output features).

4 Experimental Setup

This section elaborates upon the experiments we performed. We begin with describing the prediction model, our training and test sets and lastly, the evaluation for the model performance.

4.1 Support Vector Machine

Support vector machine (SVM) is primarily a supervised classification algorithm that can learn to separate data into two classes by drawing a hyper-plane in-between them. The coordinates of the hyper-plane are determined by ensuring maximum distance between select boundary points of the two classes and the center of the margin. The boundary points are called *support vectors*. The algorithm uses an implicit mapping of the input data to a high-dimensional feature space where the data points become linearly separable. This mapping is defined by a kernel function, i.e., a function returning the inner product between two points in the suitable feature space.

Recently, SVM has been upgraded to even perform *regression*. This is done by using a different kernel function—the ε-insensitive loss function. Basically, this function determines the regression coefficients by ensuring that the estimation errors lie below or equal to a threshold value, ε. For more information, we refer the reader to a tutorial on the topic [22].

Besides the kernel function, it is also possible to choose the SVM's kernel. In a pilot study (predicting SLOC), we found that the *linear* kernel overwhelmingly outperforms other kernels including *polynomial, radial bias,* and *sigmoid* when using the evaluation criteria presented in Section 4.3. Hence, we chose to use the same kernel across all our experiments.

4.2 Procedure

The SVM regression model learns using training data instances. For this, we randomly sample 66.67% of the data described in Section 3 to create the training set, while the remaining instances of the data (33.33%) that comprise the test set. Once the model is trained on the training data, it is tested on the test data using only the input features. The output features for the test data are predicted by the model, which are then evaluated using the measures described in Section 4.3.

Additionally, to minimise sample bias, we generate 30 independent samples of training and testing sets, and perform our experiments on each of the 30 pairs.

4.3 Evaluation

We evaluate the results from the prediction model using *PredX*, a popular performance metric used in software engineering. We chose not to use other performance metrics such as *MMRE* because they have been shown to be biased [23]. *PredX* measures the percentage of predictions made that lie within $\pm x\%$ of the actual value. The larger the value of *PredX*, the higher is the prediction accuracy. Typically, x takes the values 25 and 50. We use the same values for our evaluation.

5 Results and Discussion

Figure 4 presents the results from our experiments. All metrics are presented in alphabetical order on the *y*-axis, while the *PredX* values are plotted on the *x*-axis. For each metric, we have plotted both, *Pred25* (as circles) and *Pred50* values (as triangles) from each of the thirty experimental runs. The plots are jittered [24], i.e., a small random variation has been introduced to ease observation of overlapping values on the *x*-axis.

We observe from the figure that SLOC is predicted with reasonable accuracy. *Pred25* values hover around 42% while *Pred50* values hover around 71%. Whereas, prediction results for CBO and CBOJDK are outstandingly good. The *Pred25* values for CBO hover around 72% and even higher for CBOJDK at 86%. Their *Pred50* values hover around 88% and 97% respectively. The model also predicts RFC and DIT values with reasonable accuracy. The values of *Pred25* for both these metrics hover around 51–54%. *Pred50* for DIT hover around 77%, while the same for RFC hovers around 83%.

The prediction accuracy for other metrics, i.e., CA, LCOM, NOC, and NPM is relatively lower. Nearly all *Pred25* and *Pred50* values for most of these metrics are lower than 50%. One metric that markedly stands out is number of children (NOC). This is primarily because of the distribution of the metric. Recall from Table 2 that the median value of NOC is zero and nearly 84% files have no children. This explains the poor results for NOC.

Overall, the prediction accuracy derived from our approach is good for most metrics. It is obvious that early and reliable estimation of SLOC places projects at a vantage point by contributing substantially to their likelihood of success. Our results for SLOC demonstrate the value of our approach. Perhaps, these can be even topped by using more varied data and other prediction models.

Equally worthy is the approach's capability of predicting code-related metrics as early as during the design phase. Values of many of the metrics could be predicted with high accuracy, up to *Pred50* = 97%. The results warrant the use of our approach to facilitate many decisions pertaining to complexity, quality, and maintainability, and allow assessment of alternatives designs. If our results can be replicated in different environments, we anticipate the approach to be valuable support tool for practitioners.

6 Threats to Validity

Although we examined 89 ECLIPSE plug-ins that covered a wide spectrum of domains, from compilers to user-interfaces, we cannot claim with certainty that these plug-ins are representative of all kinds of software projects.

We also approximated the design of plug-ins by its import directives at release time. These relations may have undergone a series of changes from the initial design.

Lastly, we did not filter outliers from our data set. While doing so may improve the prediction accuracy of the models, we chose to preserve the outliers in the data since they make interesting cases to examine and realistically assess the power of our prediction models.

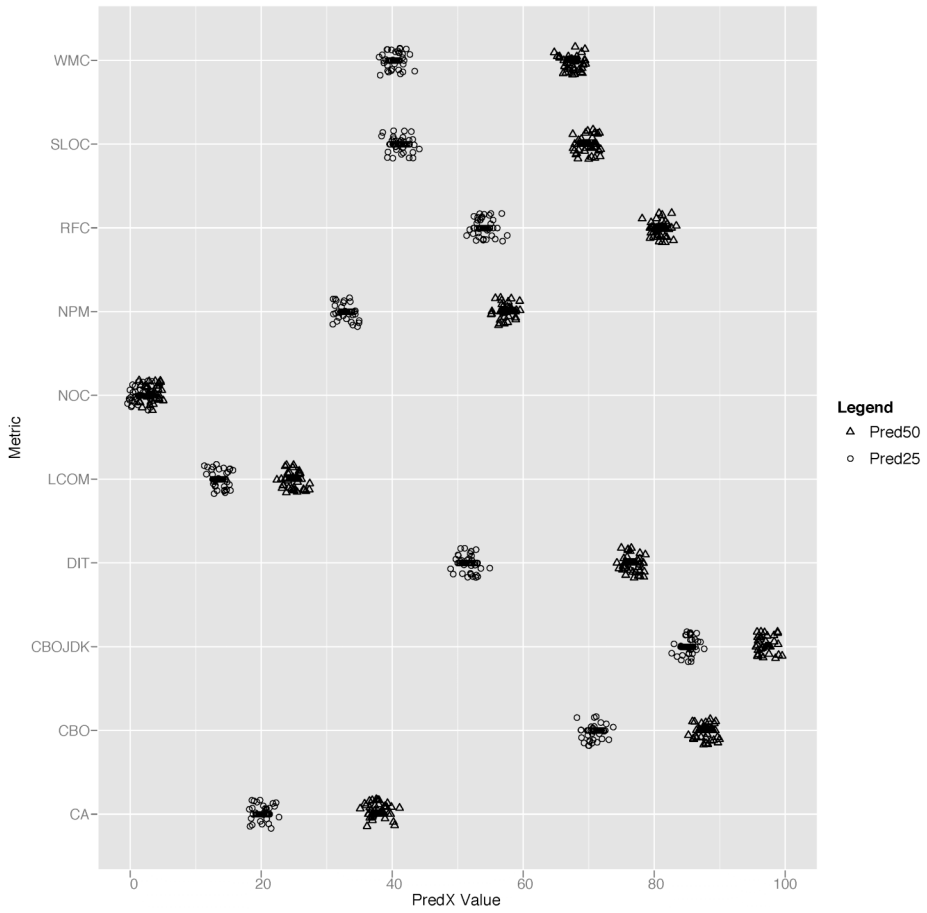

Fig. 4. Prediction accuracy for output metrics

7 Conclusions and Consequences

When it comes to components, you are what you import. As soon as you know which components you will interact with, one can already predict the future size of the component or its complexity. This allows for early estimation and allocation of resources, reducing the risk of low quality or late delivery. Even if the present study limits itself to just one platform (i.e., ECLIPSE plug-ins), the technique can easily be replicated and evaluated on other code bases.

Our approach is easily generalisable to other metrics. Most interesting in this aspect is *cost*: If we know the actual development cost of a component, we can again relate this cost to its domain—and come up with a predictor that directly predicts

development cost based on a given set of imports. Instead of development cost, we could also learn from and predict *maintenance costs* or *risk*. We are currently working to acquire appropriate data and look forward to apply our technique on it.

What is it that makes a specific domain impact software metrics? Obviously, the imports we are looking at are just a symptom of some underlying complexity—a complexity we can describe from experience, but which is hard to specify or measure a priori. Why is it that some domains require more code to achieve a particular goal? Is there a way to characterize the features that impact effort? Why do some domain result in more complex code? How do characteristics of imported components impact the features of the importers?

All these questions indicate that there is a lot of potential to not only come up with better predictors, but also to increase our general understanding of what makes software development easy, and what makes it hard. With the present work, we have shown how to infer such knowledge for specific projects—and hopefully provided a starting point for further, more general, investigations.

In addition, we have made the data set used for this study publicly available for experimentation. The data set can be accessed from the PROMISE repository at

`http://promisedata.org/`

For more information about our research on the prediction of code features visit

`http://www.st.cs.uni-sb.de/softevo/`

Acknowledgments.
Many thanks are due to the anonymous PROFES reviewers for their helpful suggestions on an earlier revision of this paper.

References

[1] Jorgensen, M., Shepperd, M.J.: A systematic review of software development cost estimation studies. IEEE Transactions on Software Engineering **33**(1) (2007) 33–53
[2] Delany, S.J.: The design of a case representation for early software development cost estimation. Master's thesis, Stafford University, U.K. (1998)
[3] Schröter, A., Zimmermann, T., Zeller, A.: Predicting component failures at design time. In: Proceedings of the 5th International Symposium on Empirical Software Engineering. (September 2006) 18–27
[4] Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A.: Predicting vulnerable software components. In: Proceedings of the 14th ACM Conference on Computer and Communications Security. (October 2007)
[5] Wheeler, D.A.: SLOCCount user's guide. http://www.dwheeler.com/sloccount/sloccount.html Last accessed 2007-11-23.
[6] Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Transactions on Software Engineering **20**(6) (June 1994) 476–493
[7] Boehm, B.: Software Engineering Economics. Prentice Hall (1981)
[8] Putnam, L.H., Myers, W.: Measures for excellence: reliable software on time, within budget. Yourdon Press, Englewood Cliffs, N.J. (1991)

[9] Mendes, E., Kitchenham, B.A.: Further comparison of cross-company and within-company effort estimation models for web applications. In: IEEE METRICS, IEEE Computer Society (2004) 348–357
[10] Shepperd, M.J., Schofield, C.: Estimating software project effort using analogies. IEEE Transactions on Software Engineering **23**(11) (1997) 736–743
[11] Kirsopp, C., Mendes, E., Premraj, R., Shepperd, M.J.: An empirical analysis of linear adaptation techniques for case-based prediction. In Ashley, K.D., Bridge, D.G., eds.: ICCBR. Volume 2689 of Lecture Notes in Computer Science., Springer (2003) 231–245
[12] Mendes, E., Mosley, N., Counsell, S.: Exploring case-based reasoning for web hypermedia project cost estimation. International Journal of Web Engineering and Technology **2**(1) (2005) 117–143
[13] Mendes, E.: A comparison of techniques for web effort estimation. In: ESEM, IEEE Computer Society (2007) 334–343
[14] Marques, M.: Eclipse AST Parser. http://www.ibm.com/developerworks/opensource/library/os-ast/ Last accessed 2008-01-14.
[15] Basili, V., Briand, L., Melo, W.: A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering **22**(10) (October 1996) 751–761
[16] Alshayeb, M., Li, W.: An empirical validation of object-oriented metrics in two different iterative software processes. IEEE Transations of Software Engineering **29**(11) (2003) 1043–1049
[17] Ronchetti, M., Succi, G., Pedrycz, W., Russo, B.: Early estimation of software size in object-oriented environments: a case study in a CMM level 3 software firm. Technical report, Informatica e Telecomunicazioni, University of Trento (2004)
[18] Aggarwal, K.K., Singh, Y., Kaur, A., Malhotra, R.: Empirical study of object-oriented metrics. Journal of Object Technology **5**(8) (2006)
[19] Subramanyam, R., Krishnan, M.: Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. IEEE Transactions on Software Engineering **29**(4) (2003) 297–310
[20] Andersson, M., Vestergren, P.: Object-oriented design quality metrics. Master's thesis, Uppsala University, Uppsala, Sweden (June 2004)
[21] Thwin, M.M.T., Quah, T.S.: Application of neural networks for software quality prediction using object-oriented metrics. Journal of Systems and Software **76**(2) (2005) 147–156
[22] Smola, A.J., Schölkopf, B.: A tutorial on support vector regression. Statistics and Computing **14** (August 2004) 199–222
[23] Foss, T., Stensrud, E., Kitchenham, B., Myrveit, I.: A simulation study of the model evaluation criterion MMRE. IEEE Transactions on Software Engineering **29**(11) (November 2003) 985–995
[24] Chambers, J.M., Cleveland, W.S., Kleiner, B., Tukey, P.A.: Graphical Methods for Data Analysis. Wadsworth (1983)