External Code Quality Model and Cross-Validation of the Model

Lech Madeyski, Zbigniew Huzar

Institute of Applied Informatics, Wroclaw University of Technology,
Wyb.Wyspianskiego 27, 50370 Wroclaw, POLAND
Lech.Madeyski@pwr.wroc.pl, Zbigniew.Huzar@pwr.wroc.pl

Abstract. One goal of this paper is to empirically explore the relationships between existing object-oriented (OO) structural measures (both, class-level and package-level) and external code quality in the context of different development methods (solo/pair programming and test-first/test-last programming) and based on 122 projects. It appeared that external code quality can be better predicted based on OO measures (class-level CBO measure, package-level NOT measure), and size related $LOCC$ measure than development methods. About 37 per cent of the external code quality variance can be explained by the model based on only aforementioned three measures. The second goal is to answer the question how accurate can these models be considering the unavoidable differences that may exist across projects and systems. This paper attempts to answer this question by means of cross-validation of the model. Two-thirds of the projects (81 projects) were randomly picked to build our prediction model and the remaining one-third (41 projects) were used to verify the efficacy of the built model. Generalizability of the model, as currently captured by existing measures, seems to be limited as about 19 per cent of the variance can be explained by the model. Therefore, possible explanations and improvements are suggested.

1 Introduction

Test-first programming (TFP) [1, 2] (often called test-driven development) and pair programming (PP) [3] have recently gained a lot of attention as key software development practices of eXtreme Programming (XP) methodology [2]. Test-first programming is a software development practice, where programmers write tests before production code, while the key aspect of pair programming is that two programmers work together, collaborating on the same development tasks. The aim of both practices, described in Section 1.2, is to improve software development process and products.

Researchers and practitioners have reported numerous, often anecdotal and favourable studies of XP practices. Empirical studies often concern productivity [4–11]. A few studies have focused on pair programming or test-first programming, as practices to remove defects [5, 12–14], influence external code quality (measured by the number of functional, blackbox test cases passed) [10, 11, 15],

or reliability of programs (a fraction of the number of passed tests divided by the number of all tests) [9, 16]. Janzen [17] has pointed out that there was no research on the broader efficacy of test-first programming, nor on its effects on internal design quality, outside a small pilot study [18]. Recently, Madeyski found that package level design quality indicators (namely package dependencies in an object-oriented design) were not significantly affected ($p > .05$) by using test-first, instead of classic (test-last) programming, or pair programming, instead of solo programming [19]. Additionally, he revealed that the number of acceptance tests passed (a measure of external code quality used by George and Williams [10, 11]) was lower when test-first programming was used, instead of test-last programming, in case of solo programmers ($p = .028$) and in pairs ($p = .013$) [15]. Moreover, there was no difference in the number of acceptance tests passed when pair programming was used, instead of solo programming [15]. Hulkko and Abrahamsson [20] also suggested that pair programming might not necessarily provide so extensive quality benefits as suggested in literature. Some of the key findings from empirical studies, concerning the impact of pair programming or test-first programming on software process and products, are summarized in tables 1 and 2.

Table 1. Summary of some empirical studies investigating pair programming practice.

Study	Environment	Subjects	Key findings
Nosek [4]	Industry	15(5P/5S)	P completed a task more quickly than S. 40% effort overhead and 30% speedup is associated with P. P achieved higher readability&functionality of products than S.
Cockburn et al. [6]	Acad./Ind.	unknown	15% effort overhead is associated with P. P improves design quality, reduces defects, reduces staffing risk, enhances technical skills, improves team communications. P is more enjoyable (statistically significant).
Williams et al. [5]	Academic	41(14P/13S)	Effort overhead associated with P is between 60% (1st assignment) and 15%(2nd&3rd assignment). 20% – over 40% speedup is associated with P.
Nawrocki et al. [7]	Academic	21(5P/5+6S)	Almost no difference in development time. 60% effort overhead and 20% speedup is associated with P.
Müller [16]	Academic	37(10P/17S)	P did not produce more reliable code than S whose code was reviewed.
Hulkko et al. [20]	Acad./Ind.	4x(4–6)	P did not provide extensive quality benefits.
Nawrocki et al. [8]	Academic	25(5P/ 5SbS/5S)	50% effort overhead is associated with P. 20% effort overhead is associated with Side-by-Side (SbS) programming.
Madeyski [15, 19]	Academic	188 (28CS/28TS/ 31CP/35TP)	There was no difference in $NATP$(Number of Acceptance Tests Passed) between solo programmers and pairs. Package dependencies were not significantly affected by using P instead of S.

Abbreviations: S(Solo programmers), P(Pairs), T(TFP, test-first approach), C(Classic, test-last approach) e.g. CS(solo programming, using test-last approach), CP(pair programming, using test-last approach), TS(solo programming, using test-first approach), TP(pair programming, using test-first approach)

Table 2. Summary of some empirical studies investigating test-first programming practice.

Study	Environment	Subjects	Key findings
Müller et al. [9]	Academic	19(9CS/10TS)	T neither produces more reliable code nor accelerates the implementation.
George et al. [10, 11]	Industrial	24(6CP/6TP)	TP products passed 18% more tests than CP. TP took 16% more time to develop an application than CP.
Williams et al. [13] Maximilien et al.[14]	Industrial	13(5CS/9TS)	Minimal or no difference in *LOC* per person-month. T reduced defect rate by approximately 40%.
Erdogmus et al. [21]	Academic	24(11TS/13CS)	TS on average wrote more tests than CS. Students who wrote more tests tended to be more productive.
Madeyski [15, 19]	Academic	188 (28CS/28TS/ 31CP/35TP)	TS passed significantly less acceptance tests than CS ($p = .028$). TP passed significantly less acceptance tests than CP ($p = .013$). Package dependencies were not significantly affected by using T instead of C.

Abbreviations: S(Solo programmers), P(Pairs), T(TFP, test-first approach), C(Classic, test-last approach) e.g. CS(solo programming, using test-last approach), CP(pair programming, using test-last approach), TS(solo programming, using test-first approach), TP(pair programming, using test-first approach)

Subramanyam and Krishnan state that research on metrics for object-oriented software development is limited, and empirical evidence, linking the object-oriented methodology and project outcomes is scarce [22]. Even more scarce is empirical evidence of the effect of pair programming and test-first programming approach on object-oriented structural complexity metrics. Therefore, Madeyski [23, 19] filled this gap and provided empirical evidence of the impact of test-first programming and pair programming on some structural complexity metrics, as design aspects are extremely important to produce high quality software [22]. However, there is no empirical evidence on relationships between structural complexity metrics and external code quality in the context of different development methods described in Section 1.2. Therefore, the aim of this paper is to fill this gap and provide external code quality model, as well as its cross-validation.

1.1 Structural Complexity Metrics

There is a statement, attributed to Kitchenham, that quality is hard to define, impossible to measure, and easy to recognize. However, this should not keep us from studying quality aspects. Therefore, we used as the quality indicators Chidamber and Kemerer (CK) structural complexity metrics [24], later refined by the same authors [25]. CK metrics are widely used, and have been validated as software quality indicators by several researchers e.g. [26], [27], [28], [29], [22] and therefore form the core of our study. CK metrics are defined as follows [25]:

– Weighted Methods per Class (WMC) measures the complexity of an individual class. If, based on [25], we consider all methods of a class to be equally complex, WMC is simply the number of methods defined in each class. This approach is commonly adopted for the sake of simplicity, and in order not

to be somewhat arbitrary, since it is not fully specified in the metric suite [26].
- Coupling Between Object (CBO) classes is the number of classes to which a class is coupled. Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class.
- Depth of Inheritance Tree of a class (DIT) is defined as the maximum inheritance path from the class to the root class.
- Number Of Children of a Class (NOC) is the number of immediate subclasses of a class.
- Response For a Class (RFC) is the number of methods that can potentially be executed in response to a message received by an object of that class.
- Lack of Cohesion in Methods ($LCOM$) is the number of pairs of methods in the class without shared instance variables, minus the number of pairs of methods in the class with shared instance variables. The metric is set to zero, whenever the above subtraction is negative.

The quality of an object-oriented design is strongly influenced by a system's package relationships. Loosely coupled and highly cohesive packages are qualities of good design. Therefore to investigate the impact of test-driven development and pair programming on object-oriented design we used Martin's package level dependency metrics [30, 31] that can be used to measure the quality of an object-oriented design in terms of the interdependences between the packages of that design. Designs which are highly interdependent tend to be rigid, unreusable and hard to maintain [30].

Martin's metrics, investigated in this study and measured by our tool [32], are defined as follows [30]:

- Afferent Couplings (Ca) — The number of classes outside the package that depend upon classes within the package.
- Efferent Couplings (Ce) — The number of classes inside the package that depend upon classes outside the package.
- Instability (I) — The ratio ($Ce/(Ca+Ce)$) of efferent coupling (Ce) to total coupling ($Ce+Ca$). This metric is an indicator of the package's resilience to change and has the range $[0, 1]$. $I = 0$ indicates a maximally stable package. $I = 1$ indicates a maximally instable package.
- Abstractness (A) — The ratio of the number of abstract classes to the total number of classes in package. This metric range is $[0, 1]$. 0 means concrete package and 1 means completely abstract package.
- Normalized Distance from Main Sequence (Dn) — This is the normalized perpendicular distance of the package from the idealized line $A + I = 1$. This metric is an indicator of the package's balance between abstractness and stability. Dn metric's results are within a range of $[0, 1]$. A value of zero indicates perfect package design.

Underlying theory about a relationship between the object-oriented metrics and fault-proneness as well as maintainability due to the effect on cognitive complexity has been provided in [33] and [28].

1.2 Development Methods

Pair programming is a practice in which two programmers (called the driver and navigator) work together at one computer, collaborating on the same development tasks (e.g. design, test, code). The driver is typing at the computer, or writing down a design. The navigator observes the work of the driver, reviews the code, proposes test cases and considers the implementation strategic implications [5, 34]. All production code is written by two people sitting at one machine [2]. In the case of solo programming, all activities are performed by one programmer.

Test-first programming is a practice based on specifying piece of functionality as a low level test, before writing production code, implementing the functionality, so that the test passes, refactoring (e.g. removing duplication) and iterating the process. The tests are run frequently, while writing production code. In the case of classic, test-last programming, tests are specified after writing production code and less frequently [21]. Test-first and test-last development activities are shown on figures 1 and 2.

As a result, four different treatments have been defined:

- Solo programming, using classic, test-last programming approach — tests after implementation (CS).
- Solo programming, using test-first programming (TS).
- Pair programming, using classic, test-last programming approach — tests after implementation (CP).
- Pair programming, using test-first programming (TP).

1.3 Problem Statement

The following definition determines the foundation for the experiment [35]:
Object of study. The objects of study are software development projects.
Purpose. The purpose is to build a model of relationships between external code quality and structural complexity metrics or development methods (CS, TS, CP, TP), described in Sections 1.1 and 1.2
Quality focus. The quality focus is the evaluation of model's goodness of fit (R^2) and its cross-validity (generalizability).
Perspective. The perspective is from the researcher's point of view.
Context. The study is based on 122 software development projects conducted in academic environment.

2 Experiment Planning

The planning phase of the experiment can be divided into seven steps: context selection, hypotheses formulation, variables selection, selection of subjects, experiment design, instrumentation and validity evaluation [35].

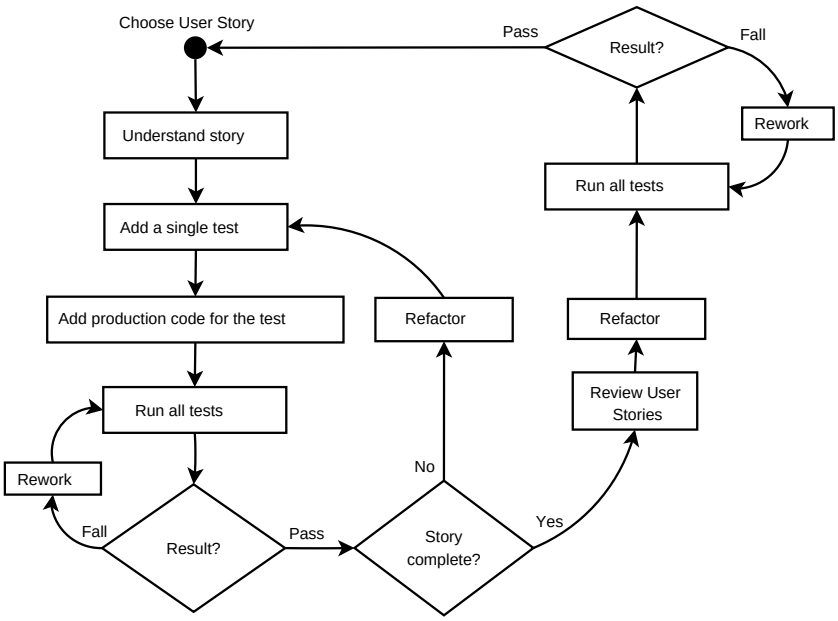

Fig. 1. Test-First Programming activities

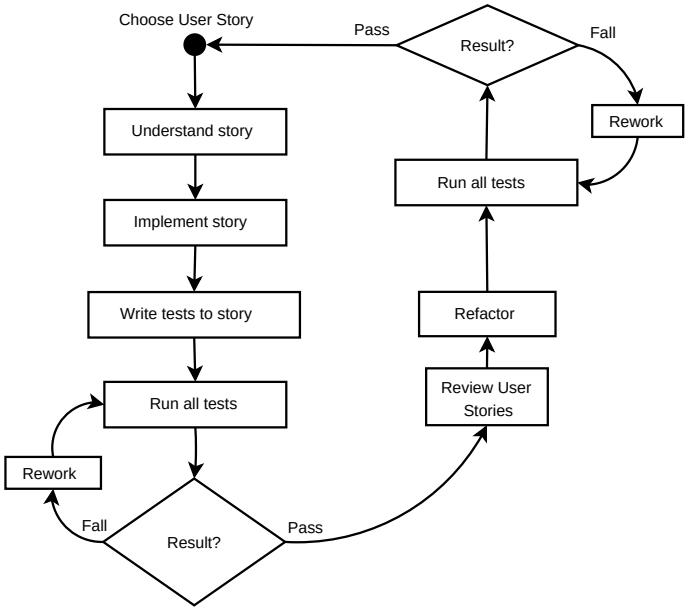

Fig. 2. Test-Last Programming activities

2.1 Context Selection

The context of the experiment was the Programming in Java course, and hence the experiment was run off-line (not industrial software development) [35]. Java was the programming language, Eclipse 3.0 was the IDE (Integrated Development Environment). All the subjects had prior experience, at least in C and C++ programming (using object-oriented approach). The Programming in Java course consisted of seven lectures and fifteen laboratory sessions (90 minutes each). The course introduced Java programming language using test-first programming and pair programming as the key XP practices. The subjects' practical skills in programming in Java, using pair programming and test-first programming, were evaluated during the first seven laboratory sessions. The experiment took place during the last eight laboratory sessions. The problem (development of the finance-accounting system) was as close to a real one as it is possible in academic environment. The requirements specification consisted of 27 user stories. The subjects participating in the study were mainly second and third-year (and few fourth and fifth-year) computer science MSc students (MSc programme of Wroclaw University of Technology is a 5-year programme). In total 188 subjects were involved in the experiment, see Table 3.

Table 3. The context of the experiment

Context factor	ALL	CS	TS	CP	TP
Number of projects:	122	28	28	31	35
Number of MSc students:	188	28	28	62	70
– in the 2nd year	108	13	16	40	39
– in the 3rd year	68	12	11	18	27
– in the 4th year	10	3	0	3	4
– in the 5th year	2	0	1	1	0
Mean value of programming experience (in calendar years)	3.8	4.1	3.7	3.6	3.9

2.2 Materials

The instruments [35] and materials for the experiment were prepared in advance, and consisted of requirements specification (user stories), pre-test and post-test questionnaires, Eclipse project framework, detailed description of software development approaches (CS, TS, CP, TP), duties of subjects, instructions how to use the experiment infrastructure (e.g. CVS Version Management System), and examples (e.g. sample applications developed using test-first programming approach, and JUnit tests). Chidamber and Kemerer, as well as Martin's metrics were collected using aopmetrics tool [32] developed and supported by members of e-Informatyka development team at Wroclaw University of Technology.

3 Impact of Test-First Programming and Pair Programming on Structural Complexity Metrics

Madeyski [19, 23] analysed the impact of test-first programming and pair programming on structural complexity metrics based on 122 projects. Madeyski [19] found that package level design quality indicators (namely package dependencies in an object-oriented design i.e. Martin's metrics) were not significantly affected by development method (CS, TS, CP, TP). Madeyski [23] also concluded that using test-first programming, instead of classic, test-last programming approach, had a significant positive impact on the mean values of RFC and CBO, two important structural complexity metrics, in the case of solo programmers, as well as in pairs. It is worth to note that CBO and RFC were also found to be good indicators of the logical stability of classes, which means there is likelihood that the class will not be change-prone, as a consequence of changes made to other classes in the design [36]. Basili, Briand and Melo [26] have shown that CBO and RFC are significantly correlated with faulty classes in multivariate analysis. As a result (as in case of CS and CP development methods using classic, test-last programming approach), the higher the CBO and RFC, the more rigorous the testing and debugging should be. It appeared that the mean value of RFC was significantly lower when test-first programming approach was used, instead of test-last programming one, in the case of solo programmers (Mann-Whitney CS vs. TS test significance value $p = .024$) and pairs (Mann-Whitney CP vs. TP test significance value $p = .031$). It also appeared that the mean value of CBO was significantly lower when test-first programming approach was used, instead of test-last programming approach, in the case of solo programmers (Mann-Whitney CS vs. TS test significance value $p = .001$) and pairs (Mann-Whitney CP vs. TP test significance value $p = .019$). There were no significant differences in the mean values of RFC and CBO when pair programming was used, instead of solo programming, as well as in the mean values of other CK metrics (DIT, NOC and WMC) between groups using different development methods. The obtained results suggest the positive impact of test-first programming practice on the two structural complexity metrics (RFC and CBO) which are also important indicators of external qualities, such as fault-proneness, but do not support the same impact of pair programming practice.

A possible mechanism, by which test-first programming practice reduces complexity of an object-oriented design, as well as coupling between objects, is that classes that have a large number of methods, calls to external methods and interconnections with other classes, would be difficult to test, and therefore, test-first programming reduces complexity of an object-oriented design, as well as coupling between objects.

An alternative explanation is that test-first programming practice slow down development [15] and thus implemented functionality, and consequently structural complexity is reduced. In fact, external code quality measure ($NATP$) can be also a measure of implemented functionality [15]. Relationships between structural complexity metrics and external code quality in the context of different development methods (CS, TS, CP, TP) will be further explored in Section 4.

4 External Code Quality Regression Model

4.1 Descriptive Statistics

Table 4 presents the descriptive statistics for the gathered measures. Columns "Max.","75%", "Med.", "25%", "Min.", "Mean", and "s.d." state for each measure the maximum value, interquartile ranges, median, minimum, mean value, and standard deviation.

Table 4. Descriptive statistics for the gathered measures

Measure (Mean)	Max	Median (Mdn)	Min	Mean (M)	Std.Deviation (SD)
$LOCC_{Mean}$	163.36	67.98	21.08	68.85	21.25
RFC_{Mean}	16.64	7.00	1.92	7.35	2.89
CBO_{Mean}	2.55	1.23	.50	1.27	.44
NOC_{Mean}	.56	.00	.00	.06	.14
DIT_{Mean}	1.19	.29	.00	.35	.24
WMC_{Mean}	11.18	6.38	3.33	6.36	1.79
$LCOM_{Mean}$	64.43	5.82	.13	8.41	9.63
Dn_{Mean}	1.00	1.00	.00	.95	.16
I_{Mean}	1.00	.00	.00	.05	.16
Ce_{Mean}	2.25	.00	.00	.15	.47
Ca_{Mean}	4.50	.00	.00	.26	.85
A_{Mean}	.17	.00	.00	.00	.02
NOT_{Mean}	32.00	9.00	2.20	9.34	4.14

Table 4 presents the descriptive statistics for the gathered measures. Distributions of the values of the measures show that inheritance has been used sparingly within the eight systems (i.e., low mean values and median for DIT, NOC).Similar results have also been found in other studies [37, 25, 38, 39]. There is, however, sufficient variance in all the measures to proceed with the analysis.

4.2 Principal component analysis

In this section, we present the results from the principal component analysis. All measures with sufficient variance (six or more non-zero data points) were subjected to an orthogonal rotation. A total of 13 measures was used.

In the correlation matrix, we found correlation coefficients of .3 and above. We also checked that the Kaiser-Meyer-Olkin measure of sampling adequacy (KMO) value is above .6 (actually it is .619), and the Barlett's test of sphericity value is significant (.000), therefore principal component analysis is appropriate.

Now we determine how many components (factors) to extract. Using Kaiser's criterion, we are interested only in components that have an eigenvalue of 1 or more. To determine how many components meet criterion we need to look in the Table 5.

Table 5. Total Variance Explained (Extraction Method: Principal Component Analysis)

Compo-	Component Initial Eigenvalues			Extraction Sums of Squared Loadings		
nent	Total	% of Variance	Cumulative %	Total	% of Variance	Cumulative %
1	4.232	32.552	32.552	4.232	32.552	32.552
2	3.133	24.101	56.653	3.133	24.101	56.653
3	1.630	12.539	69.192	1.630	12.539	69.192
4	1.154	8.879	78.071	1.154	8.879	78.071
5	.810	6.232	84.303	.	.	.
6	.704	5.419	89.722	.	.	.
7	.481	3.702	93.424	.	.	.
8	.342	2.629	96.053	.	.	.
9	.229	1.764	97.817	.	.	.
10	.205	1.575	99.392	.	.	.
11	.060	.460	99.852	.	.	.
12	.019	.148	100.000	.	.	.
13	.000	.000	100.000	.	.	.

The eigenvales for each component are listed. Only the first four components recorded eigenvalues above 1 (4.232, 3.133, 1.630, 1.154). These four components explain a total of 70.071 per cent of the variance.

Unfortunately, using the Kaiser criterion, often you will find that too many components are extracted. Fortunately, Horn's parallel analysis helps to solve the problem. If the eigenvalue is larger than the criterion value from parallel analysis, then you retain this factor; if less, then you reject it. The results are summarised in Table 6.

Table 6. Comparison of eigenvalues from principal components analysis and the corresponding criterion values obtained from parallel analysis

Component number	Actual eigenvalue from PCA	Criterion value from parallel analysis	Decision
1	4.232	1.593	accept
2	3.133	1.433	accept
3	1.630	1.311	accept
4	1.154	1.222	reject

The final table we need to look at is the component matrix, see Table 7.

Most of the items load quite strongly (above .3) on the first 2 or 3 components, while very few items load on component 4. This supports our conclusion from the parallel analysis to retain only three factors for further investigation.

Factor rotation and interpretation Once the number of factors has been determined, the next step is to try to interpret them. To assist in this process

Table 7. Component Matrix

	Component			
	1	2	3	4
Ce_{Mean}	.868	-.436	.	.
Ca_{Mean}	.812	-.384	.	.
I_{Mean}	.799	-.483	.	.
Dn_{Mean}	-.774	.532	.	.
NOT_{Mean}	-.608	.	.487	-.418
WMC_{Mean}	.516	.729	.	.
$LOCC_{Mean}$	.473	.693	.	.
RFC_{Mean}	.572	.646	.380	.
$LCOM_{Mean}$	.323	.537	.	.
A_{Mean}	.	-.407	.394	.
NOC_{Mean}	-.327	-.501	.618	.
CBO_{Mean}	.404	.	.537	-.654
DIT_{Mean}	.	-.355	.524	.593

the factors are 'rotated' by Varimax rotation, and Oblimin rotation (which does not assume that the factors are uncorrelated).

Component 1 now explains 30.14 per cent of the variance, component 2 explains 23.88 per cent, and component 3 explains 15.17 per cent. The total variance explained (69.19 per cent) does not change after rotation.

In rotated component matrix (Table 8) the main loading variables on each of the component help to identify the nature of the underlying latent variable represented by each component.

Table 8. Rotated Component Matrix (Extraction Method: Principal Component Analysis. Rotation Method: Varimax with Kaiser Normalization.)

	Component		
	1	2	3
Ce_{Mean}	.969		
I_{Mean}	.937		
Dn_{Mean}	-.934		
Ca_{Mean}	.894		
RFC_{Mean}		.935	
WMC_{Mean}		.922	
$LOCC_{Mean}$		.823	
$LCOM_{Mean}$		.642	
NOC_{Mean}			.814
DIT_{Mean}			.620
NOT_{Mean}	-.528		.558
A_{Mean}			.548
CBO_{Mean}	.326	.378	.460

The main loadings on component 1 are package level metrics (Ce_{Mean}, I_{Mean}, Dn_{Mean}, Ca_{Mean}, NOT_{Mean}). Dn is an indicator of the package's balance between abstractness and stability. Dn metric's results are between 0 and 1. A value of zero indicates perfect package design. Therefore, it is no wonder that Dn_{Mean} is negatively correlated with external code quality. The main items on component 2 are size metrics (RFC_{Mean}, WMC_{Mean}, $LOCC_{Mean}$). The main items on component 3 are inheritance metrics (NOC_{Mean}, DIT_{Mean}).

The output from Oblimin rotation is component correlation matrix (Table 9). This shows the strength of the relationship between the three factors. In our case the correlations between components are quite low, so we would expect very similar results from the Varimax and Oblimin rotation.

Table 9. Component Correlation Matrix (Extraction Method: Principal Component Analysis. Rotation Method: Oblimin with Kaiser Normalization.)

Component	1	2	3
1	1.000	.180	.064
2	.180	1.000	-.176
3	.064	-.176	1.000

We do not consider the PCs for use as independent variables in the prediction model [39]. Although this is often done with least-square regression, in the context of logistic regression, this has shown to result in models with a sub-optimal goodness of fit (when compared to models built using the measures directly), and is not current practice. In addition, principal components are always specific to the particular data set on which they have been computed, and may not be representative of other data sets. A model built using principal components is likely not to be applicable across different systems.

However, still it is interesting to interpret the results from regression analyses in the light of the results from PCA, e.g., analyse from which PCs the measures that are found significant stem from. This shows which dimensions are the main drivers of external code quality and may help explain why this is the case.

4.3 Univariate logistic regression analysis

In this subsection, we investigate the relationship of the individual measures to external code quality. The results of the univariate analysis are as follows. The measures RFC_{Mean}, CBO_{Mean}, and $LOCC_{Mean}$ are very significant ($p < .001$), while WMC_{Mean}, $LCOM_{Mean}$ are significant at $\alpha = .05$.

4.4 Correlation with size

In this section we analyze the correlation of the gathered measures to the size of the class. We use $LOCC$, the number of non-commented lines of code based

on AspectJ Abstract Syntax Tree (AST), thus independent from a style of code typing. We do this as $LOCC$ is showing strong relationship with external code quality. Spearman's rho is below .5 for the most of these measures (CBO_{Mean}, NOC_{Mean}, DIT_{Mean}, $LCOM_{Mean}$, Dn_{Mean}, I_{Mean}, Ce_{Mean}, Ca_{Mean}, A_{Mean}, NOT_{Mean}), which indicates that the correlation to size is at best moderate. Spearman's rho is above .5 for RFC_{Mean} and WMC_{Mean}.

4.5 Multiple linear regression model

The main goal is to build accurate prediction model for external code quality by using all the design measures available. We evaluate the accuracy of the prediction model we found using cross validation process and discuss generalization of such a model.

We build a model allowing all coupling, cohesion, inheritance measures (included in CK metrics suite), dependency measures (included in Martin's metrics suite), size measure ($LOCC$) and development method (CS, CP, TS, TP) to enter the model, following the forward selection process.

Obtained model (Table 10) consists of three measures: one class-level CK metric (CBO), one package-level Martin metric (NOT), one size metric ($LOCC$). It means that external code quality can be better predicted based on above measures than development methods. R^2 is a measure of variance in the dependent variable that is accounted for by the model built using the predictors. R^2 is a measure of the fit for the given data set.

Table 10. Model Summary of all 122 projects

Model	R	R Square	Adjusted R Square	Std.Error of the Estimate
1 (CBO_{Mean})	.502	.252	.246	8.291
2 (CBO_{Mean}, $LOCC_{Mean}$)	.570	.325	.314	7.910
3 (CBO_{Mean}, $LOCC_{Mean}$, NOT_{Mean})	.606	.367	.351	7.693

The $R^2 = .367$ i.e. 36.7 per cent of the variance can be explained by the model. Additionally we present the adjusted R^2 measure. The $AdjustedR^2 = .351$. The adjusted R^2 explains for any bias in the R^2 measure by taking into account the degrees of freedom of the independent variables and the sample population, i.e. R^2 values keep increasing if we add more variables. The adjusted R^2 eliminates that bias regarding the number of variables.

Models for projects developed according to CS, CP, TS, TP approach are presented below. We build CS, TS, TP models following the forward selection process and CP model following backward selection process.

4.6 Model cross-validation and conclusions

Let us further investigate our prediction model for external code quality. The accuracy of this simple model seems to be not bad. However, it is somewhat

Table 11. CS Model Summary

Model	R	R Square	Adjusted R Square	Std.Error of the Estimate
1 (CBO_{Mean})	.383	.147	.114	9.692

Table 12. CP Model Summary

Model	R	R Square	Adjusted R Square	Std.Error of the Estimate
1 (All measures)	.587	.344	-.093	10.305
...				
11 (WMC_{Mean}, RFC_{Mean})	.414	.172	.112	9.286

Table 13. TS Model Summary

Model	R	R Square	Adjusted R Square	Std.Error of the Estimate
1 (CBO_{Mean})	.631	.399	.376	6.346
2 (CBO_{Mean}, $LOCC_{Mean}$)	.712	.507	.468	5.858
3 (CBO_{Mean}, $LOCC_{Mean}$, NOT_{Mean})	.763	.582	.530	5.507

Table 14. TP Model Summary

Model	R	R Square	Adjusted R Square	Std.Error of the Estimate
1 (RFC_{Mean})	.642	.412	.395	6.832
2 (RFC_{Mean}, NOT_{Mean})	.710	.504	.473	6.377

optimistic since the prediction model was applied to the same data set it was derived from, i.e., we were assessing the goodness of fit of the model. To get an impression of how well the model performs when applied to different data sets, i.e., its prediction accuracy, we performed a cross validation of the model. This is an important step in generalization. In a regression model there are two main methods of cross validation: adjusted R^2 or data splitting, in which the data are split randomly in to parts. The adjusted R^2 value indices the loss of predictive power and tells us how much variance in dependent variable would be accounted for if the model had been derived from the population from which the sample was taken. The adjusted $R^2 = .351$ was derived using Wherry's equation.

In order to further assess the ability of the regression models to predict external code quality we use the technique of data splitting [40]. That is, we randomly pick two-thirds of the projects (81 projects) to build our prediction model and the remaining one-third (41 projects) to verify the efficacy of the built model.

The correlation between predicted and actual scores for smaller sample is squared ($R^2 = .433^2 = .187$) to compare with $R^2 = .461$ for a larger sample. As a result, the cross-validation sample is not better predicted than the sample that generated the equation. This is a usual result, but suggest limited generalizability of the model.

5 Conclusions

Our main goal was to perform a comprehensive empirical validation of the OO measures on large set of 122 projects conducted at Wroclaw University of Technology. We wanted to understand their interrelationships, and their capability to predict external code quality. To do so, a repeatable, complete analysis procedure is presented and can help the comparison of results coming from different data sets.

Many of the coupling, cohesion, and inheritance class-level measures, as well as package-level measures studied in this paper appear to capture similar dimensions in the data. In fact, the number of dimensions actually captured by the measures is much lower than the number of measures itself. This simply reflects the fact that many of the measures are somewhat redundant.

Univariate analysis results have shown that many coupling and inheritance measures are strongly related to external code quality e.g. RFC_{Mean}, CBO_{Mean}, $LOCC_{Mean}$, WMC_{Mean}, $LCOM_{Mean}$. On the other hand, inheritance, as currently captured by existing measures (DIT and NOC), does not seem to have a significant impact on external code quality.

Multivariate analysis results show that by using small subset of the measures, rather accurate models can be derived to predict external code quality (e.g. 36.7 per cent of the variance can be explained by the model that consists of only three measures i.e. CBO, NOT, $LOCC$). Unfortunately, generalizability of the model seems to be rather limited at the moment. Therefore, the crucial question is how to improve the model. Two possible improvements should be considered. First,

more sophisticated design quality metrics should be used. Therefore, aopmetrics tool [32] is being extended to include new metrics (e.g. Li metrics or variants of existing CK metrics). Second, different kind of metrics should be used. Actually, the idea is to include metrics related to the quality of tests e.g. mutation score indicator [41]. The preliminary version of mutation testing tool (called Judy) has been developed [42].

It is also possible that external code quality as a system-level quality attribute can be better predicted by system-level measures rather than structural measures of system's components (classes and packages).

6 Acknowledgments

The author expresses his gratitude to the students participating in the research, the teaching assistants and the members of the e-Informatyka team (Michał Stochmiałek, Wojciech Gdela, Tomasz Poradowski, Jacek Owocki, Grzegorz Makosa, Mariusz Sadal) for their help during preparation of the experiment infrastructure. This work has been financially supported by the Ministry of Education and Science, as a research grant 3 T11C 061 30 (years 2006-2007).

References

1. Beck, K.: Test Driven Development: By Example. Addison-Wesley (2002)
2. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change. 2nd edn. Addison-Wesley (2004)
3. Williams, L., Kessler, R.: Pair Programming Illuminated. Addison-Wesley (2002)
4. Nosek, J.T.: The Case for Collaborative Programming. Communications of the ACM **41**(3) (1998) 105–108
5. Williams, L., Kessler, R.R., Cunningham, W., Jeffries, R.: Strengthening the Case for Pair Programming. IEEE Software **17**(4) (2000) 19–25
6. Cockburn, A., Williams, L.: The Costs and Benefits of Pair Programming. In: Extreme Programming Examined. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001) 223–243
7. Nawrocki, J.R., Wojciechowski, A.: Experimental Evaluation of Pair Programming. In: ESCOM '01: European Software Control and Metrics, London, UK (2001) 269–276
8. Nawrocki, J.R., Jasiński, M., Olek, L., Lange, B.: Pair Programming vs. Side-by-Side Programming. In Richardson, I., Abrahamsson, P., Messnarz, R., eds.: Software Process Improvement. Volume 3792 of Lecture Notes in Computer Science., Springer (2005) 28–38
9. Müller, M.M., Hagner, O.: Experiment about test-first programming. IEE Procedings-Software **149**(5) (2002) 131–136
10. George, B., Williams, L.A.: An Initial Investigation of Test Driven Development in Industry. In: SAC '03: Proceedings of the 2003 ACM Symposium on Applied Computing, ACM (2003) 1135–1139
11. George, B., Williams, L.A.: A structured experiment of test-driven development. Information and Software Technology **46**(5) (2004) 337–342

12. Williams, L.: The Collaborative Software Process. PhD thesis, University of Utah (2000)
13. Williams, L., Maximilien, E.M., Vouk, M.: Test-Driven Development as a Defect-Reduction Practice. In: ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering, Washington, DC, USA, IEEE Computer Society (2003) 34–48
14. Maximilien, E.M., Williams, L.A.: Assessing Test-Driven Development at IBM. In: ICSE '03: Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society (2003) 564–569
15. Madeyski, L.: Preliminary Analysis of the Effects of Pair Programming and Test-Driven Development on the External Code Quality. In Zieliński, K., Szmuc, T., eds.: Software Engineering: Evolution and Emerging Technologies. Volume 130 of Frontiers in Artificial Intelligence and Applications. IOS Press (2005) 113–123
16. Müller, M.M.: Are Reviews an Alternative to Pair Programming? Empirical Software Engineering **9**(4) (2004) 335–351
17. Janzen, D.S.: Software Architecture Improvement through Test-Driven Development. In: OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2005) 222–223
18. Kaufmann, R., Janzen, D.: Implications of Test-Driven Development: A Pilot Study. In: OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2003) 298–299
19. Madeyski, L.: The Impact of Pair Programming and Test-Driven Development on Package Dependencies in Object-Oriented Design - An Experiment. In Münch, J., Vierimaa, M., eds.: Product Focused Software Process Improvement. Volume 4034 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer (2006) 278–289
20. Hulkko, H., Abrahamsson, P.: A Multiple Case Study on the Impact of Pair Programming on Product Quality. In: ICSE '05: Proceedings of the 27th International Conference on Software Engineering, New York, NY, USA, ACM Press (2005) 495–504
21. Erdogmus, H., Morisio, M., Torchiano, M.: On the Effectiveness of the Test-First Approach to Programming. IEEE Transactions on Software Engineering **31**(3) (2005) 226–237
22. Subramanyam, R., Krishnan, M.S.: Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. IEEE Transactions on Software Engineering **29**(4) (2003) 297–310
23. Madeyski, L.: An empirical analysis of the impact of pair programming and test-driven development on CK design complexity metrics. Technical Report PRE I31/05/P-004, Institute of Applied Informatics, Wroclaw University of Technology (2005)
24. Chidamber, S.R., Kemerer, C.F.: Towards a Metrics Suite for Object Oriented Design. ACM SIGPLAN Notices **26**(11) (1991) 197–211
25. Chidamber, S.R., Kemerer, C.F.: A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering **20**(6) (1994) 476–493
26. Basili, V.R., Briand, L.C., Melo, W.L.: A Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering **22**(10) (1996) 751–761
27. Li, W., Henry, S.: Object Oriented Metrics that Predict Maintainability. Journal of Systems and Software **23**(2) (1993) 111–122

28. Emam, K.E., Melo, W.L., Machado, J.C.: The Prediction of Faulty Classes Using Object-Oriented Design Metrics. Journal of Systems and Software **56**(1) (2001) 63–75
29. Alshayeb, M., Li, W.: An Empirical Validation of Object-Oriented Metrics in Two Different Iterative Software Processes. IEEE Transactions on Software Engineering **29**(11) (2003) 1043–1049
30. Martin, R.C.: OO Design Quality Metrics: An Analysis of Dependencies (1994, accessed September 2006)
31. Martin, R.C.: Agile Software Development, Principles, Patterns, and Practices. Prentice Hall (2004)
32. Aopmetrics: http://www.e-informatyka.pl/sens/Wiki.jsp?page=Projects.AOPMetrics (accessed September 2007)
33. Briand, L.C., Wüst, J., Ikonomovski, S.V., Lounis, H.: Investigating Quality Factors in Object-Oriented Designs: an Industrial Case Study. In: ICSE '99: Proceedings of the 21st International Conference on Software Engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1999) 345–354
34. Williams, L.A., Kessler, R.R.: All I really need to know about pair programming I learned in kindergarten. Communications of the ACM **43**(5) (2000) 108–114
35. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, Norwell, MA, USA (2000)
36. Elish, M.O., Rine, D.: Investigation of Metrics for Object-Oriented Design Logical Stability. In: CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, Washington, DC, USA, IEEE Computer Society (2003) 193–200
37. Chidamber, S.R., Darcy, D.P., Kemerer, C.F.: Managerial use of metrics for object-oriented software: An exploratory analysis. IEEE Transactions on Software Engineering **24**(8) (1998) 629–639
38. Cartwright, M., Shepperd, M.: An empirical investigation of an object-oriented software system. IEEE Transactions on Software Engineering **26**(8) (2000) 786–796
39. Briand, L.C., Wüst, J., Daly, J.W., Porter, D.V.: Exploring the relationships between design measures and software quality in object-oriented systems. Journal of Systems and Software **51**(3) (2000) 245–273
40. Briand, L.C., Wüst, J.: Empirical Studies of Quality Models in Object-Oriented Systems. Advances in Computers **59** (2002) 97–166
41. Madeyski, L.: On the Effects of Pair Programming on Thoroughness and Fault-Finding Effectiveness of Unit Tests. In Münch, J., Abrahamsson, P., eds.: Product Focused Software Process Improvement. Volume 4589 of Lecture Notes in Computer Science., Springer (2007) 207–221
42. Madeyski, L., Radyk, N.: Judy mutation testing tool project (accessed September 2007) http://www.e-informatyka.pl/sens/Wiki.jsp?page=Projects.Judy.