# Assessing and Interpreting Object-Oriented Software Complexity with Structured and Independent Metrics

Roland Neumann*

Dennis Klemann†

Hasso-Plattner-Institute for Software Systems Engineering GmbH
at the University of Potsdam, Germany

**Abstract:** Object-oriented software complexity is difficult to assess due to its manifold influences from cognition science or algorithmic complexity theory. A practical process for a structured complexity assessment is presented in this paper. It starts with considerations for measurement and data preparation. Using mathematical transformation techniques, independent complexity metrics are gained. With these results, complexity aspects of a software system can be defined. This makes a complexity comparison through system classes possible, which helps getting an overview on large systems. These process steps are then illustrated with an industrial example.

## 1 Introduction and Metrics Selection

Software functions increasingly replace specialized hardware in embedded systems. Since these functions grow in numbers, the importance for assuring quality rises additionally. The main quality aspect is correctness (prevention of faults). Faults are human based due to a possible lack of understanding of the complex software structure.

Cognitive complexity in this context means the difficulty to understand the system's behaviour, even when all system parts are known [Nor89]. The decomposition into complexity aspects enables deeper insights. A possible aspect structure for object-oriented software has been presented in [Neu04], which is used in this paper.

These complexity aspects first have to become measurable through application of software metrics. Software metrics assign numbers to various system properties. They have to comply to certain criteria [Wal90] for further mathematical inspections [Zus91]. The most important one is the reproducibility, assuring the same metrics value independent of the method or the time of measurement. To measure complexity aspects, at least one metric explaining every aspect has to be chosen. Metrics and their meanings for object-oriented software can be found in [LK94] or [Whi97].

This paper describes a practical way proposing process steps for assessing complexity

*roland.neumann@hpi.uni-potsdam.de

†dennis@klemann.org

aspects. Using these aspects, system classes can be compared and selected for further inspection leading to a better system overview. The first step in assessing complexity is measurement [BEG00] and preparation of data, which is described in the next section. Techniques for generating independent variables for assessing complexity are presented in the third section. It also includes interpretation possibilities for defining each complexity aspect. The fourth section presents an industrial example using these techniques, followed by conclusions giving prospects and a paper overview.

## 2 Preparation of Empirical Metrics Data in Practice

There are many software tools generating object-oriented software measures. When using these tools, these measures might have missing or erroneous data rows or values for various reasons (e.g. faulty measurement algorithm) in practice. Therefore empirical research starts with inspection of raw data before usage and conclusions drawn. Techniques to overcome these problems [SW99] are described in this section, as the following mathematical analysis steps are designed for complete data sets in most cases. Missing data labels either values that are completely absent or can be clearly seen as invalid (like a negative LOC count). E.g. a missing value (interpreted as Zero) decreases an average value in the data row. This influences all further techniques, like standard deviation or correlation.

### 2.1 Techniques to Deal with Missing Data

The easiest way is just to skip all data rows with missing data (called *List Deletion* - LD) [SEM01]. This may result in an immense loss of information, as there typically are comparatively few data rows in a software project. List Deletion also affects several properties of the data like mean value or correlation, which results in a biased data with a general loss of significance for further analyzes. This applies especially when the data loss is not stochastic, like no measurement at project start or for ”bad” classes.

There are more sophisticated *Missing Data Techniques* (MDT) to cope with the problem of missing data without losing that much information or changing characteristics of the data (all techniques [MSO01]). Which technique serves best greatly depends on the relation of metrics and data rows and the pattern of the missing data. The imputed number should be as low as possible since all imputed values are synthetic and may differ from the original.

The most basic MDT, *Mean Imputation* (MI), replaces gaps with the mean value of the according variable. As a result, the standard deviation of the output data set is significantly decreased. Additionally, if the chance of a missing value is related to the value itself, this technique will add a severe bias to the data set. As the imputation inserts constant values only within a metric, it will possibly weaken existing correlations to other metrics.

*Regression Imputation* (RI) does a regression analysis of the available data and fills in predicted values from the regression. This method depends on the ability to find a fitting

regression for the data. If it cannot be found, the results will greatly vary from the values that would have been originally measured. As the regression will most likely include some kind of correlation between the metrics, RI might influence the correlation of the imputed data set.

*Similar Response Pattern Imputation* (SRPI) fills the missing data with the value of the nearest data row, found by the least-squares method. This implies a certain relation between the different metrics, as the missing value is solely imputed by the other metrics. Thus, this technique increases the correlation between the metrics. This should be considered if the following steps focus on correlation.

Furthermore, this technique has the disadvantage that, in case of multiple missing values in one data row, the chosen nearest data set may be in reality not anywhere near if the original data row would be complete. This gets more severe the less metrics are measured.

### 2.2 Results after Data Preparation

The methods mentioned above have the additional drawback that it is not possible to decide whether a value is original or imputed by solely looking at the resulting data set. A way to circumvent this problem is called *Multiple Imputation*. Here, several filled data sets get imputed, using a randomized algorithm, e.g. filling in a value of this variable from a random other data row. Assuming that the further processing steps involve an appropriate tool, it is possible to spot the imputed values as well as the underlying uncertainty.

Some of these techniques leave gaps after application. In this case, these data rows can be left out (LD). The resulting quality loss will be considerably lower than before. After having applied a fitting MDT, one can use the imputed data set for the next processing steps.

## 3 Factorization: Making the Metrics Independent

Having metric values for all complexity aspects as described in the first section, and gained complete data sets from the last section, there is still an upcoming problem when drawing complexity conclusions. The interrelation between data rows of metrics (*correlation*) prevents good conclusions [CG93]. When your survey points to some high complexity values due to large class sizes, these classes cannot be separated into an inheritance structure without increasing communication between classes. Thus, drawing conclusions from correlated metrics is difficult. Techniques to remove these correlations are described below.

### 3.1 Techniques for De-Correlating Variables

There are several multivariate techniques available that fulfill the reduction of variables: *Discriminant Analysis, Cluster Analysis, Factor Analysis, Principal Components Analysis*.

*Cluster analysis* tries to partition the data set into clusters. As the original variables are affected by a multitude of underlying factors, this method is not suitable for our needs. *Discriminant analysis* reduces the variable set, but only concentrates on the "most important" variables. Thus, the underlying structure does not get clear at all. With *Factor analysis* (FA), the explanation for variances is not as focused on the first factor/component as this is the case with *Principal Components Analysis* (PCA). Because FA is concerned with explaining common variance and not total variance [Dun89], we prefer PCA over FA as uncorrelated variables were also an important requirement.

### 3.2 Using Principal Component Analysis

PCA tries to identify the underlying components of a variable set using a covariance or correlation analysis of the (normalized) original data set. Completely correlated variables are removed. Afterwards, new variables (*principal components*) are computed[1] in a way that the first component accounts for the maximum variance of the data set. In an n-dimensional space, it points in direction of maximum variance from all metrics. The next component maximally accounts for the remaining variance and so on. Each component is defined by an Eigenvector describing the influence from the original variables (with *factor loadings*), and an Eigenvalue explaining the accounting for the total variance.

The PCA also converts the data set to *Factor Scores*, which are the mappings of the measurements from the original variables to the *principal components*. Thus, through linear combination of the *Factor Scores* and the *Eigenvectors* it is possible to recalculate the original data set. Geometrically, one can imagine a variable set $(x_1, x_2, x_3)$ as the axis of a three-dimensional space. PCA finds the line (respectively the plane) where most of the measured points lie in.

These new found axes are the *Eigenvectors*, their lengths are given by their *Eigenvalues*. What PCA basically does is a rotation of the base coordinates onto the coordinate system specified by the *Eigenvectors*. The *Factor Scores* show the measured variables in the new coordinate system. An additional positive aspect of PCA besides the de-correlation is the reduction of the number of variables. The least important components (based on their *Eigenvalues*) can be neglected with minimal information loss. This decreases interpretation and further modelling effort. With the reduced, uncorrelated variable set, interpreting and further processing has been considerably simplified.

[1] A good tool support for statistical techniques is available from AddinSoft www.xlstat.com

### 3.3 Interpreting PCA Results

The resulting components are all linearly independent and thus can be interpreted separately. The analysis also points out to which extend every principal component captures the information of the original variable set. That way one can focus one's investigation on the few most important components.

To interpret the models, the meaning of the input variables has to be known. These interpretations are a prerequisite to draw conclusions from the established models. A principal component consists of a sum of metrics weighted with *factor loadings*. When adjusted with their mean, variance and the factor's Eigenvalue, they can be transformed to their percentage proportion. As the definitions of the metrics are known, the factor loadings indicate their proportion and their positive or negative influence on each factor.

In the following short example, principal component $C$ consists of 4 metrics $(LOC, DIT, NOM, CBO)$. LOC is representing the number of code lines, DIT the maximum depth of the inheritance tree to the class, NOM counts the method number and CBO the coupling count between the classes. This component $C$ is mainly formed by $LOC$ (40 % influence) and $NOM$ (50 % influence). Since these measures are size related, this component can be used as an abstract size metric. The size is dependent from the line and method count. The inheritance depth of a class hardly influences the size, but coupling (measured by CBO metric) requires invocated methods and attributes. These involve more program lines increasing the size which is reflected by its auxiliary influence.

Since the metrics are normalized, the *factor loading* can also be negative. A metric with a negative factor for a component describes a negative influence. As an example, a high metric value counting the number of classes in a system would implicate a smaller size of each class for the same functionality of the system. This would be represented by a negative *factor loading* for a class size component.

| C | LOC | DIT | NOM | CBO |
|---|---|---|---|---|
| factor loading | 1.1 | 0.2 | 0.9 | 0.4 |
| contribution [%] | 40 | 2 | 50 | 13 |

Concluding this section, the presented technique (PCA) removes interrelations constructing independent components. This reduces effort for further computation. Interpretation of these components enables drawing conclusions for possible class inspection.

The PCA and its interpretation leads to a set of independent and abstract complexity metrics. These metrics can be used for comparison of aspect values to spot outlier classes or to form complexity top lists for the whole system. This contributes to information about the most important (e.g. complex cooperating) classes for test and reengineering.

## 4 Practical Results of Factorization: Measuring Complexity

Thoughts and hypotheses of software complexity accompany software development. Functional concepts only considered structural or algorithmic complexity like McCabe's cyclomatic number [McC76] or Halsteads Software Science [Hal77]. A broader view on cognitive complexity including also communication aspects comes from theoretical approaches [CJHS92] [HK84] and recent empirical results using PCA [BW02] [BMW02]. For giving an insight into the benefits of our structured process, own empirical results are presented in this section. The empirical data comes from a component of a large commercial railway operation interlocking system. Due to its immanent risks, this system has to comply to high safety and availability requirements. Therefore there is a high need to inspect and reduce complexity in system classes.

Fifteen metrics were selected and measured based on the complexity classification scheme. Their raw data set was inspected for outliers and missing values. After usage of PCA, one metric was removed due to full correlation to another metric in this data set (no additional information). The interpreted results of complexity aspects are shown in Fig. 1 where only components with an Eigenvalue greater than 0.5 were selected, losing only $4.5\%$ data variance.

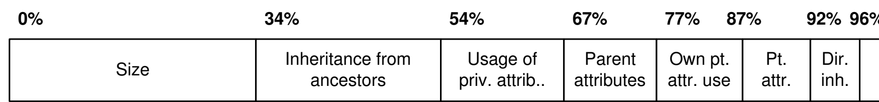


Figure 1: Complexity aspects and their proportions

The first aspect (K1) describes size related to methods, lines, public attributes. The second aspect (K2) is inheritance with its width (direct parents), depth and usage. K3 can be interpreted as usage of private attributes according to their access count and span. The usage of attributes from parent classes compared with the usage of class attributes describes K4. K5 describes the access span of protected class variables minus the further inheritance. So the class uses its protected variables itself but they are not this much used by child classes. This can be a sign for a class separable into an inheritance structure. K6 explains the access span of protected class variables with regard to inherited methods, perhaps from passing these variables as parameters to ancestor methods. The last component considered here is K7, describing the direct inheritance width to ancestor and child classes. It can be seen as multiple inheritance.

After this interpretation, prospective classes can be selected by comparing their factor scores with the majority. The interpretation of the components explains the direction of closer inspection, e.g. a large class (size related K1 greater than most classes) can be separated. Though these attributes characterize only one part of the system, they do give an impression about possible results of the presented technique. This process is concluded in the next section.

## 5 Conclusions

This paper has presented a process for structured and practical assessment of software complexity. Using this process, it is possible to generate independent complexity metrics covering important complexity aspects. With the metrics values, classes can be selected for further inspection or reengineering. The process steps include:

1. Define a complexity decomposition into adequate aspects
2. Select at least one metric to assess each aspect
3. Measure your software system using a metrics tool
4. Analyse raw data and cope with missing data
5. Perform Principal Components Analysis and interpret components
6. Compare component values through classes and assess outliers

Since this describes only a broad overview, there is still a need for a discussed and sufficient complexity classification structure. Suitability and choice of the metrics for this structure have to be proven. With empirical results of many systems, a complexity framework can be constructed to justify each new class upon. Comparison of classes between systems is a prerequisite for this framework. This enables design guidelines for early identification and prevention of complexity based faults.

## References

[BEG00] S. Benlarbi, K. ElEmam, and N. Goel.S.Rai. Thresholds for object-oriented measures. *NRC-CNRC report 43652*, 2000.

[BMW02] L.C. Briand, W.L. Melo, and J. Wüst. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE trans. Software Engineering*, 28(7):706–720, 2002.

[BW02] L.C. Briand and J. Wüst. Empirical studies of quality models in object-oriented systems. *IEEE Computers*, 56, 2002.

[CG93] R.E. Courtney and D. Gustafson. Shotgun correlations in software measure. *Software engineering journal*, pages 5–13, 1993.

[CJHS92] S.N. Cant, D.R. Jeffrey, and B. Hendersson-Sellers. A conceptual model of cognitive complexity of elements of the programming process. *TR Centre for information technology research*, (57):52–63, 1992.

[Dun89] G. Dunteman. *Principal Component Analysis*. SAGE Publications, 1989.

[Hal77] M.H. Halstead. *Elements of Software Science*. Elsevier, New York, 1977.

[HK84] S. Henry and D. Kafura. The evaluation of software system's structure using quantitative software metrics. *Software - Practice and Experience*, 14(6):561–573, 1984.

[LK94] M. Lorenz and J. Kidd. Object-oriented software metrics. *Prentice-Hall*, 1994.

[McC76] T. J. McCabe. A Complexity Metric. *IEEE Transaction on Software Engineering*, SE-2:308–320, 1976.

[MSO01] I. Myrtveit, E. Stensrud, and U. Ohlsson. Analysing data sets with missing data: An empirical evaluation of imputation methods and likelihood-based methods. *IEEE Trans. SW-Eng.*, 27(11):999–1013, 2001.

[Neu04] R. Neumann. A categorisation for object oriented software metrics in fault prediction. *Proc. Software measurement European Forum*, pages 287–296, Jan. 2004.

[Nor89] D. Norman. *Dinge des Alltags: Gutes Design und Psychologie für Gebrauchsgegenstände*. Campus, Frankfurt / New York, 1989.

[SEM01] K. Strike, K. ElEmam, and N. Madhavji. Software cost estimation with incomplete data. *IEEE Trans. Software Eng.*, 27(10):890–900, 2001.

[SW99] N.D. Singpurwalla and S. Wilson. *Statistical Methods in Software Engineering*. Springer, New York, 1999.

[Wal90] E. Wallmüller. *Software-Qualitätssicherung in der Praxis*. Hanser, München / Wien, 1990.

[Whi97] S. Whitmire. *Object-Oriented Design Measurement*. Wiley, 1997.

[Zus91] H. Zuse. *Software Complexity*. DeGruyter, 1991.