Mixing SNA and Classical Software Metrics for Sub-Projects Analysis.

ROBERTO TONELLI  
University of Cagliari  
DIEE  
P.za D’Armi, 09100 Cagliari  
ITALY  
roberto.tonelli@dsf.unica.it

GIUSEPPE DESTEFANIS  
University of Cagliari  
DIEE  
P.za D’Armi, 09100 Cagliari  
ITALY  
giuseppe.destefanis@diee.unica.it

Abstract: We present a preliminary study of the joint application of network and software metrics to the analysis of subprojects of a large software system. Aim of this paper is to provide new tools for the analysis of software systems subprojects by using very recent results obtained in the new research field of software networks. We present an empirical and exploratory analysis of the evolution of software subprojects in time, as well as a comparison among subprojects with similar or different aims and functionalities. Our preliminary results show how the joint application of traditional and network software metrics may be used to identify subprojects developed with similar functionalities and scopes.

Key–Words: Software Metrics; Social Network Analysis; Software Networks; PCA; Typing manuscripts, \LaTeX

1 Introduction

The recent litterature on software engineering shows how it is possible to describe large software systems through a complex graph [1] [2], where the graph nodes are the software modules (packages, files, classes or other software entities), and graph edges are the relationships between modules [14].

In the graphs associated to Object Oriented (OO) software systems the nodes are the classes, which are related each other through different kind of binary relationships, such as inheritance, composition and dependence.

Several authors already investigated some software networks properties, like the distribution of Fan-in or Fan-out of network nodes [6] [9] [11], finding features characteristic of complex networks [8]. The complex network structure may be thus analyzed as a Social Network, introducing metrics which are relatively new for software, but are already used in Social Network Analysis (SNA) with different meanings [10] [12]. The advantage of this approach is to provide new metrics for software systems, which describe how the classes interact with each other, and which may be related to software quality [15] [5]. Up to now, all these studies regarded a software system as a whole.

On the other hand, more traditional software metrics, like the CK-suite [3], have historically been related to software quality, and from a complete different point of view with respect to software network theory, tackle also the problem of describing interrelationships among classes.

In this paper we perform an empirical analysis of the Eclipse sub-projects with the joint application of CK metrics, SNA metrics, and other network metrics, in order to detect similarities and differences among some subprojects depending on their scope and functionalities. The joint use of the different set of metrics, allows to explore the structure of such software networks exploiting relationships existing among the sub-systems. [4]. [?].

2 The Software Systems and the Metrics Analyzed.

The Eclipse system is a standard dataset for software engineering studies which provides different sub-projects large enough for statistical significance of the results, and possesses a number of different releases which allow to analyze the time evolution of such subprojects. It is an Open Source OO software system whose source code is available for public download from the Eclipse web site. The releases analyzed are 2.0, 2.1, 3.0, 3.1, 3.2, 3.3 and 3.4. Subsequent releases are the evolutions of the previous ones, providing new functionalities while keeping the older ones. Thus, the number of sub-projects varies from the 80 of Eclipse 2.0 to the 407 of Eclipse3.4. Among all these sub-projects, some are too small to be meaningfully represented as software network and provide
degenerate values for some SNA metrics. Consequently, we analyzed only subprojects with more than 40 classes.

Furthermore, in order to carry out the analysis of the subproject’s time evolution, we study only those subsystem which belong to all the examined releases, which are 22.

For each class in the sub-systems we are faced with the following set of metrics: Locs, Fanin, Fanout, cbo, rfc, wmc, lcom, noc, dit, reachEfficiency, effSize, closeness, dwReach, infoCentrality, Size, Ties. Two are internal metrics, namely Locs and lcom. The others are network metrics, and depend on the structure of the software system, namely are computed on the software graph. The last seven are metrics taken from Social Network Analysis (SNA), and provide indications on how the classes interact with each other. Some of the SNA metrics are defined on a subgraph, the ’EGO’ network (from the Latin EGO, meaning ‘self’), which is the subnetwork obtained considering one node, named the EGO node, and only the nodes directly connected to it.

In all our software networks the links among nodes, due to the different relationships among the classes, are undirected. Thus, our software graphs are undirected graphs. The definition of each metric is the following:

- **Loc**: the number of lines of code of the class, excluding blank lines and comments. Although some authors have pointed out the deficiencies of Loc, it is still the most commonly used size measure in practice because of its simplicity.

- **Weighted Method per Class (WMC)**: the number of methods of the class, or interface. It is a measure of class complexity. Note that we set the weighting factor to one, as it is the case in most published computations of WMC. In general, it has been shown that the larger WMC, the more the development effort and the larger the probability of defect-proneness.

- **Number of Children (NOC)**: the number of immediate subclasses of the class, a CK metric.

- **Depth of Inheritance Tree (Dit)**: the number of inheritance levels, from the object hierarchy top to the given class.

- **Response for Class (RFC)**: a CK metric computed as the sum of the number of methods defined in the class, and of the cardinality of the set of methods called by them and defined in external classes.

- **Coupling Between Object Classes (CBO)**: the number of other classes the given class is coupled to. It is a CK metric that denotes class dependency on other classes in the system.

- **Lack of Cohesion of Methods (LCOM)**: the difference between the number of non-cohesive method pairs and number of cohesive pairs, a CK metric.

- **In-degree (Fan-in)**: the number of in-links of the class in the class graph, denoting how much the class is used by other classes in the system.

- **Out-degree (Fan-out)**: the number of out-links of the class in the class graph, denoting how much the class uses other classes in the system. It is strictly correlated to CBO CK metric, measuring the coupling of a class with other classes of the system.

- **Size**: Size of the EGO-network related to the considered node (i.e. Class); it is the number of the nodes of the EGO-network.

- **Ties**: Number of edges of the EGO-network related to the node.

- **EffectiveSize (effSize)**: Effective size of the EGO network; the number of nodes in the EGO network minus one, minus the average number of ties that each node has to other nodes of the EGO network.

- **reachEfficiency**: the percentage of nodes within two-step distance from a node, divided by the EGO Size.

- **Closeness**: the Closeness is the reciprocal of the Farness, where the Farness is defined as the sum of the length of all shortest paths from the node to all other nodes.

- **Information Centrality (infoCentrality)**: the harmonic mean of the length of paths starting from all nodes of the network and ending at the node, multiplied by the total number of nodes.

- **dwReach**: the sum of all nodes of the network that can be reached from the node, each weighted by the inverse of its geodesic distance. The weights are thus 1/1, 1/2, 1/3, and so on.

### 3 Data Analysis

Our set of metrics is possibly exahustive for describing software systems, but it is also cumbersome and...
presents many correlations. For example, Size = Fanin + Fanout + 1, CBO = Fanin + Fanout, Fanout is correlated to Locs since the larger the number of lines of code, the more is the likelihood for the class to present dependencies from other classes, and so on.

In order to achieve only the relevant informations carried by the metrics and to eliminate redundancies related to the metrics correlations, we performed a PCA on all the sub-projects with more than 40 classes, using as input the metrics for each class. The output is a set of measures for each class, where only few of them, which are uncorrelated, the so-called principal components, retain the original relevant information [7].

The results show that all the systems present a few main principal components, explaining the majority of the variance, and that usually more than the eighty percent of system variability is well accounted for by the first three principal components. This suggests that for describing the system only few principal components are needed. A typical situation is: pca1 = 55 %, pca1 = 18 %, pca3 = 10 %, for a cumulative percentage of 83% of the variance explained by the three first PC’s.

Using the principal components each class may be identified by means of only few coordinates which carry the relevant information contained in the original set of metrics. This reduced set allows to represent each class as a point in the PCA space, and the entire subprojects as sets of points in the PCA space. One immediate consequence is that as the software system evolves, and the subprojects number of classes grows, the number of points in the PCA space also grows. If the metrics of the existing classes change, the principal components will also change and the points representing the classes will change position in the PCA space.

The pattern of the class distribution in such PCA space is thus representative of the main structure of the metrics for each subproject.

In particular, it is possible to follow the time evolution of the sub-projects during the software development, and to easily detect major changes of the whole structure, or minor changes related to the modification of a few classes. Through this analysis it is possible to detect relationships among the patterns of different sub-projects, and to see the patterns changes as a same sub-project evolves in time.

Figs. 1 and 2 show the patterns in six releases of the sub-projects 'jtd.core' and 'pde.ui' (we did not show Eclipse 3.0 release of 'pde.ui' and Eclipse 3.3 release of 'external.ui.tool', only for reasons of space availability, being the patterns for these releases identical to those of the other ones).

The figures show that these patterns are consistent for each sub-project across the releases. This property is not trivial, since the PCA takes into account normalized values, thus even changes in a single class may influence the position of all others. On the other hand, there may be sub-projects that undergo major modifications, for example because the number of classes increases significantly from one release to another, or because they may undergo refactoring, where the code and the relationships among the classes are substantially changed.

In the first case, it is not granted that the new classes are still distributed according to the pattern exhibited by the older classes. Thus, a permanence of the same pattern when the number of classes varies significantly may be a signature of an existing structure associated to the specific software system.

In the second case, refactorings may significantly change the metrics, especially those computed on the software graph. This may be reflected in major changes of the pattern of the corresponding sub-project during the evolution from one release to another.

The situation is illustrated in the Figures 1 and 2. In Fig. 1 the project jtd.core starts with about 800 classes, and ends with about 1150. The pattern remains the same across the evolution, even if the number of classes increases. So, the new classes are arranged according to the same pattern. In Fig. 2 we show the project pde.ui, with about 370 classes in Eclipse 2.0, 480 in Eclipse 2.1, and ending with about 1020 in Eclipse 3.4. The final number of classes is about three times the initial one, still the pattern does
not change during the evolution, and is different from the jtd.core pattern. We examined also the other sub-projects finding always the same patterns through the various releases.

Figures 1 and 2 show also that the patterns are well defined for each sub-project, but are different among the two sub-projects. Consequently we decided to compare all the patterns of the 22 sub-projects belonging to the same release, finding that they are in general similar for sub-projects whose name ends with the same desinence, which is related to the project functionalities. Figures 3 and 4 show the patterns exhibited in Eclipse 3.4 by the sub-projects whose names end by `.core` and by `.ui`, respectively. The latter are projects relative to the user interface, and are conceived for providing functionalities clearly different from the `.core` projects.

We found that sub-projects developed with the same scope exhibit in general the same patterns. This is also confirmed by the analysis of the cumulative percentage of the variance related to the principal components. These percentages confirm the same findings related to the patterns. For all the `.core` projects the first PC has a minor relative weight with respect to the others, and a relatively large contribute to the variance is due also to the second PC. For all the `.ui` projects the first PC has a major relative weight with respect to the others, while the second PC provides a minor contribute to the total variance, comparable to the contribute of the remaining PCs.

Finally we reversed the analysis for understanding which original metrics contribute much to the firsts principal components, contributing to the patterns more than the other metrics. The varimax rotation and the obtained biplots, not reported for reasons of space, allow us to determine the contribution of each metric to the first two PCs, and to verify how the obtained patterns are not an artifact due to the PCA, but are directly determined by the metric values, and are different for sub-projects written for different purposes.

In particular, for the .core sub-projects, the metrics Size, Ties, effSize and Fanin, contribute roughly in the same amount to PC1 and to PC2, while metrics cbo, wmc, rfc, Locs, and Fanout, give an opposite contribute to PC2 whit respect to the previous ones. Differently, for the .ui sub-projects, the contribution of each metrics to the two PCs is not well defined.
and each metric provides a contribute to the first and second PC that is generally different from that of the other metrics.

4 Threats to Validity

First, Eclipse could not be representative of all Java systems. Further studies on different systems, open source and commercial are needed to extend the validity of our study.

Second, Eclipse is not representative of systems written in other languages. Thus, a full investigation on the possibility of using the PCA of the metrics for describing the complex network structure of the whole software system must include a comprehensive study, spanning several languages and systems.

Third, we analyzed the subset of the 22 projects belonging to all the releases. The analysis of the similarities among patterns of sub-projects with the same scope must be extended to cover also the sub-projects born in the subsequent releases. In this case one may expect that, in any case, User Interfaces sub-projects may in general keep the same patterns, since these are due to the particular values of some metrics, which, for interfaces, may be in general different. For the core sub-projects this may be not valid anymore, since they provide basic functionalities which may be different for sub-projects created since the beginning of the whole software systems, with respect to core sub-projects added along the way.

5 Conclusion

We reported a preliminary and exploratory analysis of the Eclipse subprojects, using a joint application of SNA and traditional software metrics. The entire set of metrics has been summarized performing a PCA and obtaining a very reduced number of independent principal components, which allow to represent the classes into a space where they show typical patterns. These patterns may be useful in providing information about the subprojects functionalities and their time evolution through different releases. Since the Principal Components are linear combinations of the original metrics measured for classes, the patterns are determined by these original metrics.

Our methodology may thus be useful in monitoring the evolution of subprojects in time, and may be indicative of similarities in the functionalities of some subprojects. The relationship among metrics and patterns suggests that, with the introduction of the "bugs metric" into the input set of metrics for the PCA, this methodology might eventually provide indications about software quality.

References:
