2006

Efficiency implications of open source commonality and reuse

E. Capra
capra@elet.polimi.it

Chiara Francalanci
Dipartimento di Elettronica e Informazione Politecnico di Milano, francala@elet.polimi.it

Francesco Merlo
Politecnico di Milano, merlo@elet.polimi.it

Macello Tosetti

EFFICIENCY IMPLICATIONS OF OPEN SOURCE COMMONALITY AND REUSE

Eugenio Capra, Politecnico of Milan, Department of Electronics and Information, via Ponzio 34/5, 20133 Milano (Italy), capra@elet.polimi.it

Chiara Francalanci, Politecnico of Milan, Department of Electronics and Information, via Ponzio 34/5, 20133 Milano (Italy), francala@elet.polimi.it

Francesco Merlo, Politecnico of Milan, Department of Electronics and Information, via Ponzio 34/5, 20133 Milano (Italy), merlo@elet.polimi.it

Marcello Tosetti, former student of Politecnico of Milan, Department of Electronics and Information

Abstract

This paper analyzes the reuse choices made by open source developers and relates them to cost efficiency. We make a distinction between the commonality among applications and the actual reuse of code. The former represents the similarity between the requirements of different applications and, consequently, the functionalities that they provide. The latter represents the actual reuse of code. No application can be maintained for ever. A fundamental reason for the need for periodical replacement of code is the exponential growth of costs with the number of maintenance interventions. Intuitively, this is due to the increasing complexity of software that grows in both size and coupling among different modules. The paper measures commonality, reuse and development costs of 26 open-source projects for a total of 171 application versions. Results show that reuse choices in open-source contexts are not cost efficient. Developers tend to reuse code from the most recent version of applications, even if their requirements are closer to previous versions. Furthermore, the latest version of an application is always the one that has incurred the highest number of maintenance interventions. Accordingly, the development cost per new line of code is found to grow with reuse.

Keywords: commonality, software reuse, software cost, maintenance and replacement policies, open source.

1. INTRODUCTION AND LITERATURE REVIEW

This paper presents the results of an empirical research investigating the impact of code reuse on the development costs of open source applications. The paper makes a distinction between commonality and reuse. The former represents the similarity between the requirements of different applications and, consequently, the functionalities that they provide. The latter represents the actual reuse of code. Applications with high commonality may not be characterized by a correspondingly high reuse of code. This occurs when different programs provide the same functionalities.

Clearly, the high commonality – low reuse situation is economically inefficient (cf. Morisio et al., 2002). Open source should enhance reuse, as any software house can retrieve the required source code from shared repositories. As a consequence, open source should reduce development costs (Rothenberger et al., 2002 proposes a cost-benefit-model for systematic software reuse; see also Tomer et al., 2004). The cost benefits of open source are generally accepted and broadly advocated as a driver of savings and faster response to users (cf. Fitzgerald, 2004). These opportunities are also regarded as a means to an effective reduction of the digital divide (cf. Keats, 2004). However, the literature does not provide quantitative benchmarks of the cost reductions enabled by open source. Cost reductions have been demonstrated by discussing success stories and isolated case studies, but have not been studied extensively (cf. Morad et al., 2005, Russo et al., 2003, Salmivalli et al., 2004, Madanmohan et al., 2004).

The software engineering literature states that systems can change through either vertical or horizontal evolution (Parnas, 1976). The former is due to slight changes in requirements to adapt to new business situations, while the latter is due to a differentiation of the scope of the application along its evolutionary path. Both typologies of changes are carried out through either maintenance or replacement (cf. Basili et al., 1996, Barua and Mukhopadhyay, 1989, and Belady et al., 1985). While maintenance modifies an existing application by creating a new version of the same code, replacement satisfies requirements by developing a new application from scratch. The degree to which code is actually reused represents the discriminant between maintenance and replacement (cf. Basili, 1990).

No application can be maintained for ever (cf. Lehman et al., 2000 and Visaggio, 1999). A fundamental reason for the need for periodical replacement of code is the exponential growth of costs with the number of maintenance interventions (see Section 2, and cf. Ahn et al., 2003). Intuitively, this is due to the increasing complexity of software that grows in both size and coupling among different modules. New functionalities must be integrated with existing modules and, thus, integration costs soar (cf. Tan et al., 2005 and Bianchi et al., 2001). This suggests that there exists a limit to the cost efficient reuse of open-source software.

This paper analyzes the reuse choices made by open source developers and relates them to cost efficiency. We make a distinction between the commonality among applications and the actual reuse of code. If developers make a replacement decision, code is not reused even if commonality is high. In open source contexts, this may not be an explicit business decision, but may be due to a difficulty in discovering and analyzing the required reusable code. This represents an additional cost of reuse which may further reduce benefits.

The paper provides a model of the concepts of commonality and reuse. The model supports the creation of software evolution graphs that show maintenance vs. replacement decisions. Two nodes with the same parent represent a substantial differentiation of code which splits into different applications. Conversely, two nodes along the same edge represent subsequent versions of the same application. Real software evolutionary graphs are built by analyzing open source repositories. The depth and breadth of trees, the

time and effort required to develop subsequent versions of applications, the degree of commonality and reuse are analyzed as indicators of maintenance vs. replacement decisions and cost efficiency.

The presentation is organized as follows: Section 2**Fel! Hittar inte referenskälla.** formalizes the concepts of commonality, reuse, maintenance and replacement. Section 3 presents the empirical model adopted to measure commonality, reuse and cost efficiency in a real open source context. Section 4 discusses empirical findings according to the model proposed. Finally, Section 5 draws preliminary conclusions and proposes further work to deepen the research.

2. PRELIMINARY DEFINITIONS

This section provides formal definitions of the concept upon which our analysis are based. Sub-section 2.1 introduces maintenance and replacement according to software engineering literature. Sub-section 2.2 proposes a definition of commonality and reuse oriented to the analysis of software evolution.

2.1. Definitions of maintenance and replacement

Maintenance is defined as the operation that builds a software system S_i following a change in the functional requirements of the system R_i. By definition then changes in requirements causes changes in the software system.

This definition excludes corrective maintenance, which takes place even if the set of requirements does not change ($R_{i-1}=R_i$). However, adaptive and perfective maintenance are considered, which traditional software engineering studies have found to constitute more than 75% of total maintenance effort (Basili et al., 1996, Chan et al., 1996, Lientz et al., 1981).

Replacement is defined as the operation that replaces a system S_{i-1} with a new system S_i that differs from S_{i-1} by a given percentage of code.

It is important to note that replacement can take place even if requirements do not change ($R_i=R_{i-1}$). While $R_i \neq R_{i-1} \Rightarrow S_i \neq S_{i-1}$, the opposite is not always verified.

Banker (1993) and Tan (2005) provide an expression for maintenance cost which takes into account four different components:

- fixed cost of maintenance operations, such as the effort required to plan and organize the task of implementing a new set of functionalities;
- linear cost of specifying requirements and developing new functionalities;
- quadratic cost associated with the integration of new modules of code among themselves (as suggested by Banker et al., 1993);
- cost due to the integration of new modules of code with pre-existing software, exponentially proportional to the number of interventions on the application and to the *entropy* of the system (see Bianchi et al., 2001).

The last term is related to the fact that each maintenance operation inevitably increases the "chaos" of a software system and, thus, the effort required by subsequent maintenance operations. Hence, no application can be maintained forever and replacement is needed to reset the effects of software degradation.

The software engineering and information systems literature offer several metrics to measure reuse benefits, some derived from enterprise case studies (e.g., Banker et al.,1991, Rothenberger et al.,1999), other from economic models (e.g., Poulin et al., 1993; cf. Mili et al., 1999 for a comparison of seventeen literature models of software reuse). However, the literature has not developed a model explaining the relationship between reuse and maintenance. Basili (1990) suggested that reuse can be seen as a

maintenance intervention on existing code and, therefore, if a software is reused multiple times, it is inevitably subject to degradation. As a consequence, reuse reduces the linear cost associated to development, but can increase the non linear cost related to integration, if it is not optimized. The following section proposes an empirical model of commonality, reuse and development costs to empirically analyze their relationship.

2.2. Commonality and reuse

Commonality is formally defined as an assumption held uniformly across a given set of objects (Coplien et al., 1998). If this definition is applied in software engineering, commonality becomes a property of the requirements of a software architecture that enables the reuse of pre-existing components.

Let us consider two software systems S_i and S_j. If R_i and R_j represent the requirements sets of S_i and S_j, the commonality of S_i with S_j is defined as:

$$\xi_{i,j} = \frac{\left|R_i \cap R_j\right|}{\left|R_j\right|}, \tag{1}$$

Commonality is equal to 0 if requirements are completely different and, therefore, their intersection is empty, while it is equal to 1 if requirements are identical. $\xi_{i,j}$ represents the average fraction of function points that can be re-used from S_i to implement S_j.

The **size growth of the requirements set** from S_i to S_j is defined as:

$$\delta_{i,j} = \frac{\left|R_j\right|}{\left|R_i\right|}, \tag{2}$$

If $\delta_{i,j}$ is greater than 1, S_j's requirements set is greater than S_i's and, therefore, S_j's implementation is likely to require a higher number of function points.

The following relation holds:

$$\frac{\xi_{i,j}}{\xi_{j,i}} = \delta_{j,i}, \tag{3}$$

It is important to note that definition (1) refers to requirements as opposed to code. As a consequence, the concept of commonality does not coincide with reuse. For example, identical sets of requirements, $R_i=R_j$, may be implemented by distinct modules on different platforms.

The actual **reuse** of application *i* to build application *j* is defined as:

$$\eta_{i,j} = \frac{\left|S_i \cap S_j\right|}{\left|S_j\right|}, \tag{4}$$

$\eta_{i,j}$ represents the fraction of code of S_i that can be reused to implement S_j. It can be easily verified that $\eta_{i,j}$ varies from 0, when S_j must be developed from scratch, to 1, when S_j reuses all the code of S_i. S_i

and S_j may represent two versions of the same application which has undergone maintenance or replacement.

Similar to (2), **the size growth of the code** set from S_i to S_j is defined as:

$$\sigma_{i,j} = \frac{|S_j|}{|S_i|}, \tag{5}$$

Similar to (3), the following relation holds:

$$\frac{\eta_{i,j}}{\eta_{j,i}} = \sigma_{j,i}, \tag{6}$$

3. AN EMPIRICAL MODEL OF COMMONALITY AND REUSE

This section explains how empirical data were collected to support analysis. Sub-section 3.1 describe how the sample of applications were selected. Sub-section 3.2 proposes empirical measures of commonality and reuse, which were formally introduced in Sub-section 2.2. Sub-section 3.3 presents software evolutions graphs and describes how they can be extracted from empirical data. Sub-section 3.4 describes how development costs were empirically measured.

3.1. Selection of the sample of applications

On the web, there are several open software repositories which allow free browsing and retrieval of code in various functional areas. Our empirical analysis is based on the Sourceforge.net repository. This choice is motivated by the high number of projects available from Sourceforge.net (more than 100.000) and by the completeness of project description.

Sourceforge.net classifies software projects along several dimensions, including topic, programming language, licence, operating system and status (alpha, beta, mature, inactive,…). Analyses have focused on a specific category of projects defined as follows:

- **Topic:** Databases (more than 35% of total projects), Software Development, Internet and some minor projects from Text Processing, Interpreters, Testing, Frameworks and Security;
- **Language:** Java
- **Status:** mature/ active

Topics were selected proportionally to the availability of projects which satisfy the following requirements:

- a sufficient number of applications versions have been issued (at least 5), so that evolution analysis are significant;
- description notes are available for all the applications versions and focus on the functionalities, not only on bug fixing, because our commonalities measures are based on these description notes (see Sub-section 3.2);
- source code of all applications versions is available.

Java language was selected because the tool we developed to measure reuse (see Sub-section 3.2) identifies and compare Java classes and is language-specific.

The analysis was limited to mature/active projects, i.e. projects at a mature state of development and continuously evolving, in order to exclude random-managed open code from our sample, as suggested by Rainer et al. (2005). A total of 26 projects have been selected corresponding to a total of 171 application versions. These projects do not constitute all the projects selected by the criteria presented above, but were randomly selected among them. Our empirical analysis are currently being extended and consolidated with data from more projects. However, given the limited standard variations of the interpolations which will be presented in Section 4, we believe that our sample of applications is sufficiently large to support preliminaries conclusions.

Accordingly to the selection criteria adopted, for each project the following data are available from Sourceforge.net:

- code of all application versions, constituting the history of the project;
- release date of each version;
- detailed descriptions of each version, including change logs and bug fixing reports.

These data have been used to measure model parameters, as described in the following sections.

3.2. Measures of commonality and reuse

A theoretical definition of commonality is provided by equation (1) of Section 2.1. To simplify the empirical analysis, the set of requirements R_i of version i of an application is obtained from the functional descriptions provided by Sourceforge.net. These descriptions are expressed in natural language. A list of significant words is built from text as follows:

- elimination of stop words (e.g., prepositions, conjunctions,…) and syntactical pre-processing (e.g., caps/small cap, singular/plural, tenses of verbs,…) as suggested by Castano et al. (1999) and Li et al. (2003);
- elimination of non significant words according to a context-dependent thesaurus which has been designed ad hoc (e.g., in a DBMS project the word “database” is likely to add little value to requirements).;
- construction of a table with significant words and their rate of occurrence;

Let us consider an example to clarify this type or representation. The following sentence in the description notes of an application:

“The DBMS application allows to connect online to MySQL server; concurrent queries can be managed directly by the application or through MySQL server.”

would result in Table 1:

Table 1 – Example of table extracted from the description of an application.

Significant words	**Occurrence**
MySQL	2
server	2
Connect	1
Online	1
concurrent	1
query	1

Significant words	Occurrence
directly	1
manage	1

The commonality among two versions is computed by counting the words which appear in the tables of both versions and weighing them with their occurrence rate, according to the following expression:

$$\xi_{i,j} = \frac{\sum Occ(W_k)}{\sum Occ(W_j)}, \tag{7}$$

where *Occ(W)* measures the number of occurrence of word *W* in a table and W_k are the words belonging to both the tables of applications *i* and *j*.

Reuse, which is defined by equation (4), can be measured at several levels: application level, Java class level, method level or code level. The application level seems too aggregate, while the code can be misleading, as the same constructs of a programming language can be used with different formats or syntactical rules. In this paper, reuse is measured at the Java class level. A class is considered reused if it has the same declaration as a pre-existing class.

Both commonality and reuse have been computed by means of a tool developed as part of this research.

3.3. Software evolution graphs

In Sub-section 2.2, the concepts of commonality and reuse have been used to describe the evolution of an application through its maintenance operations. This evolution has been defined as *vertical*. Through reuse, an application may also evolve into two different applications. This means that at a certain stage *i-1* of its lifetime, a set of requirements R_{i-1} changes originating two different requirement sets R_i^1 and R_i^2. These new requirement sets can either evolve trough maintenance or further split along different evolution paths. This origination of multiple requirement sets has been defined as *horizontal evolution*. This complex evolution scenario can be represented by means of a *commonality software evolution graph*, which bears some resemblance to the biological evolution tree.

In general, R_i^1 and R_i^2 will be similar to R_{i-1}, i.e. will have a high degree of commonality with their *parent* requirement set. However, the greater the distance between two requirement sets within the software evolution graph, the lower their commonality.

Similarly, a *reuse software evolution graph* can be built by considering the actual evolution of code S_i. If an application evolves with an high rate of reuse from previous versions, i.e. through maintenance interventions, the graph is linear, as S_{i+1} will have S_i as a parent. When a replacement takes place, a new branch of the graph is be generated. Intermediate cases are also possible, as a version of an application can generate two children versions, which are different from each other.

Commonality and reuse software evolution graphs have been built for all projects in our sample by applying the algorithm described below. As the method is almost the same for commonality and reuse, a description is provided only for the former.

- Matrix □ is computed, where □*(i,j)*=□$_{i,j}$. Only elements with $j>i$ are computed, as symmetry relation (3) holds for commonality.
- Projects are analyzed separately. All the applications within a project are considered one by one in a chronological order. The first application is a root.
- For each application j a parent is identified. Application i^* is defined as parent of j if and only if the following requirements are fulfilled:
  - i^* has been released before j
  - $\xi_{i^*,j} = \max_{\forall i}(\xi_{i,j})$
  - $\xi_{i^*,j} \geq \rho$, where □ is a constant which has been set to 0,7 according to empirical pilot tests
- If $\xi_{i,j} < \rho$ □i temporally antecedent to j, then j is defined as root and a new branch is generated in the graph.

Figure 1 shows an example of commonality and reuse graphs built for an open-source project. It can be noted that commonality and reuse software evolution graphs are quite different from each other. The implications of this difference is discussed in Section 4.

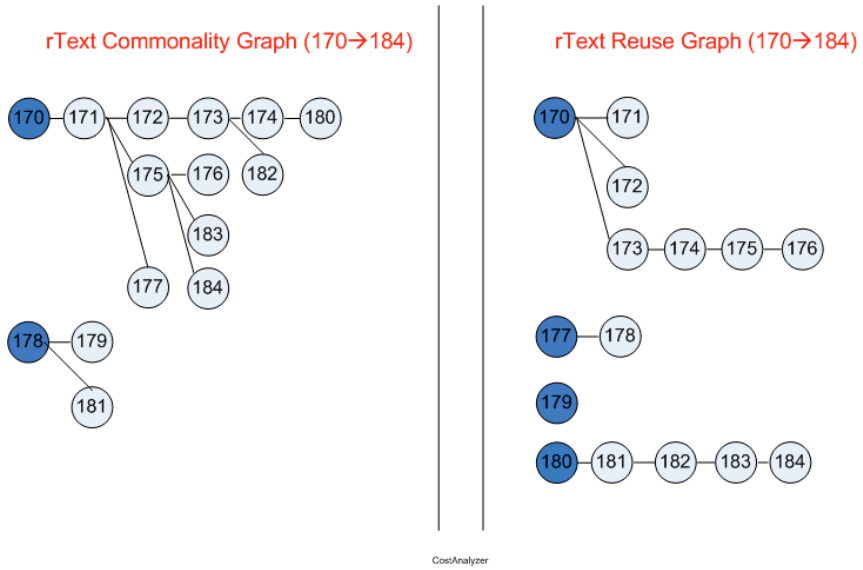

Figure 1. Example of commonality and reuse software evolution graphs.

3.4. Measures of costs

It is well known (Strike et al., 2001) that the measure of software development cost is difficult, especially for an extensive sample of applications. Given the data provided by Sourceforge.net, the time elapsed between the release of two subsequent applications within the same project is taken as a proxy of implementation cost. Such a measure has the following drawbacks:

- more than one person may be working on a project;
- developers may be working part time on a project, especially in an open-source context;
- developers may not be working on a project over the entire elapsed time between two releases.

Future work will address these drawbacks and refine our cost measure.

4. EMPIRICAL FINDINGS

Empirical analyses highlight three fundamental findings. First, *reuse evolution graphs are in general more linear* than commonality graphs. In general, developers tend to reuse from the most recent version of an application, even if a previous version has greater commonality (see Figure 1). Therefore, developers do not reuse code from the applications with the most similar set of requirements, although this may lead to higher integration costs.

In order to measure the linearity of commonality and reuse evolution graphs, we define *linearity* as follows:

$$linearity = 1 - \frac{graph\ \ entropy}{n}\ , \tag{8}$$

where n is the total number of the nodes of the graph and *graph entropy* is defined as:

$$graph\ \ entropy = (r-1) + \sum_{l \in L} l \cdot n(l)\ , \tag{9}$$

where

- r is the number of root nodes in the graph;
- $L = \{l_1, \ldots, l_n\}$ is the set of the cardinalities of edges outgoing from nodes;
- $n(l)$ is the number of nodes which have exactly l outgoing edges.

It can be noted that this metric ranges from 0 to 1 and assigns the maximum value of linearity (1) to graphs which are a linear sequence of nodes (case *a* in Figure 2), and the minimum value (0) to two-level graphs with a single root (case *b* in Figure 2) or to graphs entirely composed by roots (case *c* in Figure 2).

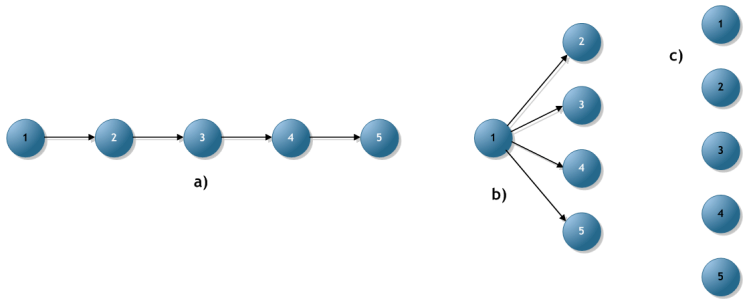

Figure 2. Sample graphs with linearity=1 (a) and linearity=0 (b, c).

Linearity has been measured for both commonality and reuse evolution graphs of our sample of applications (Section 3). Results are shown in Figure 3. The figure clearly points out that the linearity of reuse graphs is greater than the linearity of commonality graphs for all applications in our sample (all data points are located in the upper left area of the graph). This proves that development initiatives generally start from the latest release of each application, leading to non-optimized reuse strategies. This is consistent with previous literature indicating that the lack of domain analysis and repository management specific tools as a failure factor in reuse programs (Morisio et al. 2002).

The second result is that *the cost of software development decreases as reuse grows*. Evidence for this assertion is provided in Figure 4, plotting the absolute development costs of all application versions as a

function of reuse. Costs are expressed in terms of development days as explained in Section 3.4, while reuse values have been calculated as explained in Section 3.2.

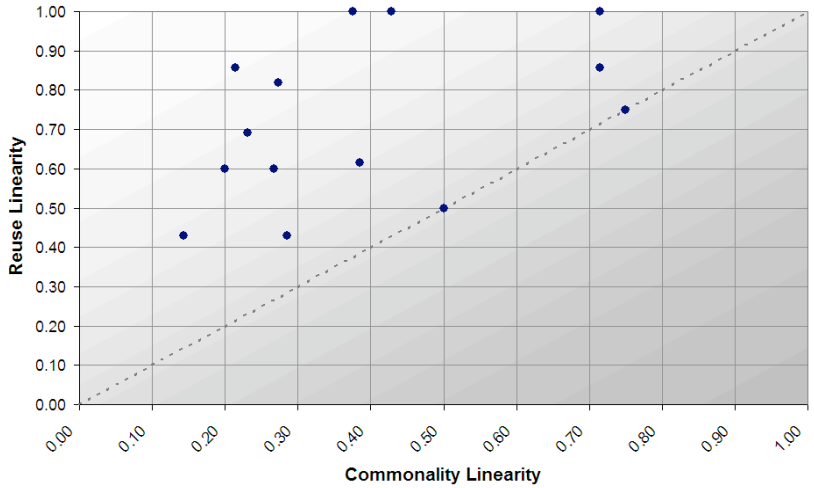

Figure 3. Linearity of reuse and commonality evolution graphs of a sample of applications (topic: Database*).*

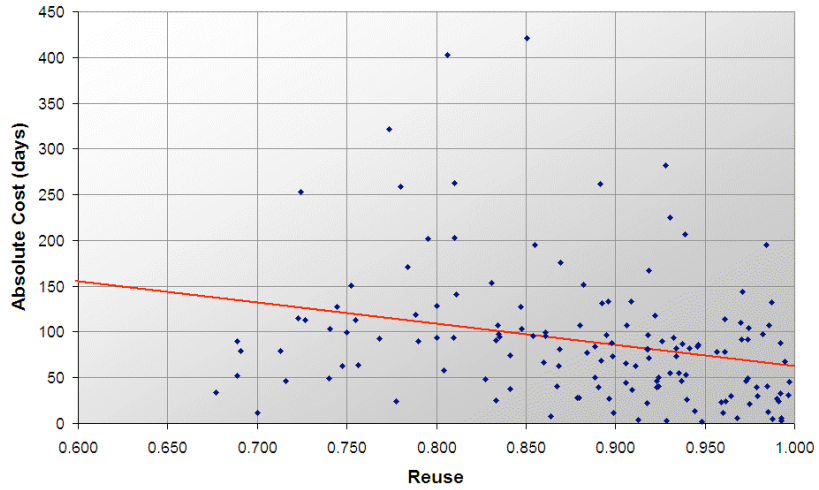

Figure 4. Absolute development costs as a function of reuse (Linear interpolation function: m=-230*;* q=293; □=±70*).*

The third and most valuable finding is that *the cost per new line of code increases with reuse*. This can be interpreted as a consequence of the growing complexity of software as the number of maintenance interventions increasing.

Given the data provided by Sourceforge.net and the measurement of actual reuse between different versions of the same application, we have measured the cost of each version in terms of days per new line of code normalized in the 0-1 range. Results are shown in Figure 5.

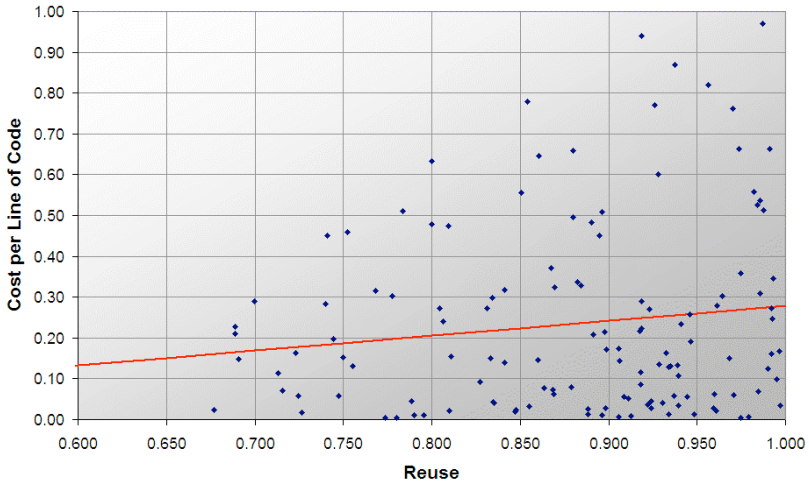

Figure 5. Cost per new line of code as a function of reuse for our sample of applications (Linear interpolation function: m=0,37*;* q=-0,09; □=±0,23*).*

The graph highlights that costs are an increasing function of reuse. The continuous line interpolates data points showing the increasing trend of cost per line of code. This proves that reusing software in a non-optimized way is not a cost-efficient strategy for software maintenance, probably due to the high cost of integration of reused lines of code (see Section 2.1).

5. DISCUSSION AND CONCLUSIONS

Results show that reuse choices in open-source contexts are not cost efficient. Developers tend to reuse code from the most recent version of applications, even if their requirements are closer to previous versions. If a previous version of an application provides code fulfilling current requirements, this code should be reused. If it is redeveloped, an implicit replacement decision is made which may not be cost efficient.

Furthermore, the latest version of an application is always the one that has incurred the highest number of maintenance interventions. Maintenance increases software complexity and, thus, development costs. Consistent with this software engineering principle, the development cost per new line of code is found to grow with reuse.

These results indicate that open source contexts lack methodologies and tools supporting design decisions. In particular, code repositories are not managed according to project needs and no decision support is provided. The lack of coordination that seems to characterize open source development makes reuse inefficient. Costs are higher than they would be if reuse was organized according to software engineering principles by taking into account requirements and related commonality.

The validity of our results obviously depend on the soundness of our measurement, which at present are only proxies of the theoretical concepts we have defined. Accordingly, future work will refine the measures of commonality, reuse and cost.

In this work, commonality is measured by means of a syntactical processor of description notes of applications. In future work it will be measured based on the concept of semantic similarity, as suggested by Batini et al. (1996) and Li et al (2003). Furthermore, description notes will be integrated with

information taken from user interfaces of Java applications. This will provide a finer description of requirements which, in turn, can be translated into a more precise measure of commonality.

Reuse is measured at the class level. This can lead to errors when refactoring interventions are carried out which deeply change the structure of a class without changing its name. In future work reuse will be measured at the method level, based on graph theory (applying techniques similar to those proposed by Grove et al.,1997).

The measure of costs will be refined by considering the number of people working on a project and their time allocation, as opposed to the elapsed time between subsequent versions. This will involve a specific data-collection effort with developers. Furthermore, the sample of considered applications will be extended to consolidate the results of the analysis.

References

Ahn Y., Suh J., Kim S. and Kim H., "The Software Maintenance Project Effort Estimation Model Based on Function Points", Journal of Software Maintenance and Evolution: Research and Practice, vol. 15, no. 2, pp. 71-85, 2003

Banker R., Datar S., Kemerer C., Zweig D., "Software Complexity and Maintenance Costs", Comm. ACM, vol. 36, no. 11, pp. 81-94, 1993

Banker R., Kauffman R. J., "Reuse and Productivity in Integrated Computer-Aided Software Engineering: An Empirical Study", MIS Quarterly, vol. 14, no. 4, pp. 420-433, 1990

Barua A., Mukhopadhyay T., "A cost analysis of the software dilemma: to maintain or to replace", Proc. of the 22th Annual Hawaii International Conference on System Sciences, vol. III: Decision Support and Knowledge Based Systems Track, pp. 89-98, Jan 1989

Basili V., "Viewing maintenance as reused-oriented software development", IEEE Software, January 1990

Basili V., Briand L., Condon S., Kim Y.-M., Melo W.L., Valett J.D., "Understanding and Predicting the Process of Software Maintenance Releases", Proc. 18th International Conference of Software Engineering (ICSE 1996), pp. 464-474, 1996

Batini C., Castano S., De Antonellis V., Fugini M.G., Pernici B., "Analysis of an Inventory of Information Systems in the Public Administrations", Requirements Eng, vol. 1, no. 1, pp. 47-62, 1996

Belady L. A., Lehman M. M., "Program Evolution", Processes of Software Change, Academic Press, New York, 1985

Bianchi A., Caivano D., Lanubile F., Visaggio G., "Evaluating Software Degradation through Entropy", Proc. IEEE Seventh International Software Metrics Symp. (METRICS 2001), pp. 210-219, 2001

Castano S., De Antonellis V., Fugini M.G., Pernici B., "Conceptual Schema Analysis: Techniques and Applications", ACM Transactions on Database Systems, vol. 23, no. 3, pp. 286-333, 1999

Chan, T., Chung S., Ho T., "An Economic Model to Estimate Software Rewriting and Replacement Times", IEEE Transaction on Software Engineering, vol. 22, no. 8, pp. 580-598, 1996

Coplien J., Hoffman D., Weiss D., "Commonality and Variability in Software Engineering", IEEE Software, November-December 1998

Fitzgerald B., "A critical look at open source", IEEE Computer, vol. 37, no. 7, pp. 92-94, 2004

Grove D., DeFouw G., Dean J., Chambers C., "Call Graph Construction in Object-oriented languages", Proc. Object Oriented Programming Systems Languages and Applications, pp. 108-124 ,1997

Keats D.W., "Addressing digital divide issues in a partially online masters programme in Africa: the NetTel@Africa experience", Proc. IEEE International Conference on Advanced Learning Technologies, pp. 953-957, Sept. 2004

Lehman M., Kahen G., Ramil J., "Replacement Decisions for Evolving Software", Proc. of 2nd Workshop on Economics Driven Software Engineering Research, 2000

Li Y., Bandar Z.A., Mclean D., "An approach for measuring semantic similarity between words using multiple information sources", IEEE Transactions on Knowledge and Data Engineering, vol. 155, no. 4, pp. 871-882, 2003

Lientz B.P., Swanson B., Software Maintenance Management, Addison-Wesley, 1981

Madanmohan T.R., Rahul De', "Open Source Reuse in Commercial Firms", IEEE Software, 2004

Lim, W.C. "Reuse Economics: A Comparison of Seventeen Models and Directions for Future Research", Proceedings of the Fourth International Conference on Software Reuse, pp. 41-51, 1996

Morad S., Kuflik T., "Conventional and open source software reuse at Orbotech – an industrial experience", Proc. IEEE International Conf. on Software–Science, Technology and Engineering, pp. 110-117, Feb 2005

Morisio M., Ezran M., Tully C. "Success and Failure Factors in Software Reuse", IEEE Transaction on Software Engineering, vol. 28, no. 4, pp. 340-357, 2002

Parnas D., "On the Design and Development of Program Families", IEEE Transactions on Software Engineering, vol. SE-2, no. 3, pp. 1-9, 1976

Poulin J.S., Caruso J.M., "A Reuse Measurement and Return on Investment Model", 2nd International Workshop on Software Reusability, Lucca, Italy, 1993

Rainer A., Galen S., "Sampling open source projects from portals: some preliminary investigations", Proc. 11th IEEE International Software Metrics Symposium (METRICS 2005),2005

Rothenberger M.A., Dooley K.J., "A Performance Measure for Software Reuse Projects", Decision Sciences, vol. 30, no. 4, 1999

Rothenberger M.A., Nazareth D., "A cost-benefit-model for systematic software reuse", Proc. 10th European Conference on Information Systems (ECIS 2002), pp. 371-377, June 2002

Russo B., Zuliani P., Succi G., "Toward an Empirical Assessment of the Benefits of Open Source Software", Proc. International Conference on Software Engineering (ICSE'03), pp. 117-120,May 2003

Salmivalli L., Nissilä J., "Curing health care information systems with open source software", Proc. 12th European Conference on Information Systems (ECIS 2004), 2004

Strike K., El Emam K., Madhavji N., "Software cost estimation with incomplete data", IEEE Transactions on Software Engineering, vol. 27, no. 10, pp. 890-908, 2001

Tan Y., Mookerjee V.S., "Comparing Uniform and Flexible Policies for Software Maintenance and Replacement", IEEE Transaction on Software Engineering, vol. 31, no. 3, March 2005

Tomer A., Golden L., Kuflik T., Kimchi E., Schach S.R., "Evaluating Software Reuse Alternatives: A Model and Its Application to and Industrial Case Study", IEEE Transaction on Software Engineering, vol. 30, no. 9, pp. 601-612, 2004

Visaggio G., “Assessing the maintenance process through replicated, controlled experiment”, Journal of Systems and Software, vol. 44, no. 3, pp. 187-197, 1999