Software Qual J (2011) 19:553–578
DOI 10.1007/s11219-010-9113-8

Benchmarking library and application software with Data Envelopment Analysis

Alexander Chatzigeorgiou · Emmanouil Stiakakis

Published online: 19 September 2010

Abstract Library software is generally believed to be well-structured and follows certain design guidelines due to the need of continuous evolution and stability of the respective APIs. We perform an empirical study to investigate whether the design of open-source library software is actually superior to that of application software. By analyzing certain design principles and heuristics that are considered important for API design, we extract a set of software metrics that are expected to reflect the improved nature of libraries. An initial comparison by conventional statistical analysis confirms the overall belief that products of different software size scale should not be compared by simply examining metric values in isolation. In this paper, we propose the use of Data Envelopment Analysis (DEA), borrowed from production economics, as a means of measuring and benchmarking the quality of different object-oriented software designs captured by software metrics and apply this approach to the comparison of library and application software. The advantages offered by DEA and the differences between the application of DEA in an economic and a technological context are discussed. Results of the approach are presented for 44 open-source projects, equally divided between libraries and applications.

Keywords Object-oriented design · Software metrics · Benchmarking · Data Envelopment Analysis · Efficiency

1 Introduction

The use of quantitative approaches for the assessment of high-level software quality attributes, such as maintainability, has a long tradition in the field of Software Engineering. According to several hierarchical quality models such as ISO 9126 (ISO 1991), quality attributes of a software system can be mapped to sub-characteristics which in turn, can be evaluated by appropriate source code metrics. However, researchers agree that metrics

A. Chatzigeorgiou (✉) · E. Stiakakis
Department of Applied Informatics, University of Macedonia,
156 Egnatia Str., 54006 Thessaloniki, Greece
e-mail: achat@uom.gr

should not be examined individually since the overall picture is provided by a number of complementary measures (Arisholm and Sjøberg 2004). As a result, when the goal is to assess and moreover to compare software products, metrics should be combined to allow valid interpretation.

All stakeholders of the software development process are interested in identifying the best software projects and measuring the performance or quality of other projects against an observed best practice frontier. However, a comparison focusing on a single metric in isolation each time entails two significant risks that threaten the validity of the analysis. Both risks are well known to any software practitioner employing software metrics for quality assurance: (a) If metrics accurately capture design properties of a software system, the overall quality is reflected by the complete set of the selected metrics, not by each one of them. In other words, the fact that a software system exhibits a superior design does not necessarily imply that it excels in all quantitative measures. (b) To compare independent metric values from software projects of completely different scales in terms of delivered functionality and behavior is like comparing apples and oranges. As an example, it would be controversial to claim that an extremely poor system in terms of functionality is better than a significantly larger one, simply because it has lower cyclomatic complexity or lower coupling.

Similar problems are encountered in economics where researchers also aim at benchmarking different companies that have varying size characteristics and whose efficiency is a non-linear function of a number of inputs and outputs. In a financial context, obviously one cannot claim that a company A is more efficient than a company B, simply because its profit is larger. It might be the case, that the larger profit of A is achieved at the cost of ten times the amount of investment. For a single input/output process, efficiency could be calculated as the ratio of output over input. However, quantifying efficiency is not as trivial when there is a non-linear relationship between inputs and outputs, and the factors affecting performance are numerous.

In this paper, we propose an alternative approach for benchmarking object-oriented software by treating the software design process as a business unit or production process (von Mayrhauser et al. 2000). In the same way that a company is interested in maximizing output (such as profit), for a given input (such as raw materials or investment), a software designer is interested in maximizing certain metrics (or in some cases minimizing others) for a given functionality to be implemented. Since the complex interactions of software production process components do not allow the analytic specification of the production function relating outputs and inputs of the software design process (von Mayrhauser et al. 2000), the proposed approach is based on Data Envelopment Analysis (Charnes et al. 1978), which involves the use of linear programming methods to measure the performance of so-called Decision Making Units (DMUs). DEA is suitable for benchmarking companies or in our case software products, because in contrast to other approaches, such as multivariate regression, that identify a theoretical baseline for comparison, DEA constructs an actual best practice frontier. In the context of software design, comparing against a theoretical and possibly not feasible design is less useful to the stakeholders of the software development process, who prefer to know about existing products that are well-designed and employ best practices. Moreover, DEA enables the comparison of systems that have varying size in terms of functionality and state, in contrast to conventional approaches based on independent metric assessment, which are only meaningful when systems with the same or similar size are compared.

As a case study for investigating the suitability of DEA as a means to compare software systems, this paper attempts to compare and benchmark open-source library and application software. In particular, we investigate whether actual open-source libraries exhibit improved design properties, as captured by appropriate metrics, compared to open-source application software. The selected metrics have been chosen by analyzing certain design principles that are believed to be important in API design. The corresponding metrics are the outputs of the software design process assuming that the designer's goal is to maximize their values (such as the abstraction level). As inputs to the design process, we consider the amount of functionality to be implemented and the size of system state captured by the number of concrete methods and the number of attributes, respectively. These inputs have been selected in analogy to a number of previous approaches that treat the design process as an optimization problem (see Sect. 5.1). In these design optimization problems, the goal is to minimize (maximize) certain functions such as coupling (cohesion). The independent variables are the behavior of the system (represented by its methods) and its state (represented by class attributes) (Bowman et al. 2007).

The proposed approach has been applied to 22 open-source applications and 22 open-source libraries. The results from DEA enable us to classify the analyzed designs based on their overall efficiency and to identify which aspects should be improved and to what extent. Although the approach is limited to a small set of metrics, the experience from using DEA in order to benchmark object-oriented designs seems promising.

The rest of the paper is organized as follows: sect. 2 discusses a set of design principles and the reasons they are important in API design and lists the corresponding metrics. Section 3 presents the results of an initial statistical analysis. A brief overview of DEA and the differences between the application of DEA in an economic and a technological context are provided in Sect. 4. The input variables to DEA, as well as the data for the projects under investigation, are presented in Sect. 5. The results of the application of normal and system differentiated DEA are given and discussed in Sect. 6, while Sect. 7 summarizes the major threats to validity. Related work is presented in Sect. 8. Finally, we conclude in Sect. 9.

2 Design principles and corresponding metrics

The software engineering literature has systematically recorded a number of design principles that should be followed when developing object-oriented systems (Martin 2003) or design heuristics that should not be violated when taking design decisions (Riel 1996). Designing a shared library is considered a far more complicated task than building in-house, closed application software (Tulach 2008). The reasons are mainly the number of clients depending on that piece of software implying backward compatibility and the need for constant evolution in a way that does not disturb clients. These requirements impose a stricter design style making the conformance to design rules even more important. That is, at least, what most programmers think of library software that is accessible through a well-defined Application Programming Interface (API). DEA is proposed in this paper to test the hypothesis that library software follows certain design principles to a larger extent than application software.

Since the selection and definition of suitable measures depends strongly on specifying clearly a measurement goal, we formulate the goal of this study according to the Goal-Question-Metric paradigm by Basili et al. (1994):

Analyze software designs
for the purpose of evaluating their conformance
with respect to generic API design guidelines

from the perspective of the researchers and developers
in the context of 44 open-source software systems.

Table 1 presents a number of design principles or design heuristics (first column) along with the reason for believing that the corresponding rule is important in API design (second column). The third column presents a quantitative measure that is strongly or loosely related to the corresponding design principle or heuristic along with a brief explanation.

Obviously, these are not all of the design principles and heuristics related to API design, which is a broad field by itself. Other properties that could be quantified and included in a study of API design quality could be the use of consistent parameter ordering across methods or cautious overloading. A thorough summary of good practices for API design can be found in (Bloch 2006). However, it should be borne in mind that the discrimination power of a DEA model (as the one that will be presented next) improves as the number of outputs becomes lower. In other words, a model that would incorporate additional metrics as outputs would not achieve a sharp discrimination among the examined projects.

The selected set of rules and metrics should be regarded as a representative sample to exemplify the use of DEA for benchmarking software systems with a focus on APIs. Emphasis is given in the approach rather than the input/output data values. As an example, it could be the case that one of the projects exhibits a low usage of the final keyword due to a specific design decision. Obviously, such a decision cannot be revealed by analyzing numbers and characterizing the corresponding project, after the application of DEA, as not efficient, might not be fair. Although the study of the reasons that cause a project to deviate from the advisable levels is beyond the scope of our work, the results of DEA might be a good starting point to perform thorough analysis on particular cases.

3 Statistical comparison

The conventional approach in comparing library and application software by means of metrics would be to treat each kind of software as a different group of data and employ statistical analysis in order to compare these groups for significant differences. Since for each of the selected metrics we have a one scale, numeric dependent variable that follows a normal distribution, divided into two unrelated groups, the primary test of choice is an independent sample *t*-test (Wohlin et al. 2000). For the data set shown in Sect. 5, the corresponding *t*-test statistic results are summarized in Table 2. The first column indicates the corresponding metric, the second column the 2-tailed significance value, and the third column the mean difference between the two groups (for the cases where the significance value is lower than 0.05). The hypothesis being tested can be stated as (Wohlin et al. 2000):

$$\mathrm{H}_0\colon \mu_{\mathrm{Library}} = \mu_{\mathrm{Application}}$$

$$\mathrm{H}_1\colon \mu_{\mathrm{Library}} \neq \mu_{\mathrm{Application}}$$

where μ_{Library} and $\mu_{\mathrm{Application}}$ is the mean of the corresponding dependent variable for libraries and applications respectively.

As it can be observed, for two of the selected metrics the significance level does not allow us to draw any conclusions, regarding the superiority of library or application software as reflected by the metric values. For the metrics where the significance is lower than 0.05, the fact that the abstraction level and percentage of final fields are higher for APIs and that coupling (MPC) is lower for APIs agrees with our initial belief concerning API design practices.

Table 1 Important design principles for APIs and corresponding measures

Design principle/design heuristic	Reason for being important in APIs	Corresponding measure
Dependency inversion principle—DIP “High-level modules should not depend on low-level modules. Both should depend on abstractions” (Martin 2003)	APIs should define immutable contracts to ease client programming (Tulach 2008). If clients depend on unstable modules, they become volatile when the libraries evolve. According to the dependency inversion principle, API clients should depend on abstractions which are both stable and allow extensions to functionality by subclassing. Put in another way, clients (and especially API clients), should code against interfaces, not implementations (Gamma et al. 1995). The stability of interfaces limits the amount of changes that can be propagated to clients causing maintainability problems.	Conformance to the corresponding principle is expected to be (at least partially) reflected on the system’s level of abstraction, that is the number of abstract classes and interfaces over the total number of classes. *Metric*: Abstraction level $= \frac{\#abstract\ Classes + \#\text{interfaces}}{\#classes}$
Liskov substitution principle—LSP “Subtypes must be substitutable for their base types” (Liskov 1988; Martin 2003)	Subclassing API classes opens a far wider range of class usage beyond the API designer’s original intention. Inheritance can violate encapsulation: certain uses can violate the Liskov substitution principle destroying the validity of the involved objects or causing further problems such as violations of the open-closed principle (Martin 2003). This risk is particularly intense when clients extend base classes without having full knowledge of their internals, as is the typical case of API usage. For the sake of future evolution and to limit the number of ways in which an API can be used, subclassing should be disallowed by making either all concrete classes final or at least by making most methods final (Tulach 2008).	The corresponding metrics simply refer to the extent by which “final” is employed in the declaration of classes and methods. *Metrics*: Percentage of final concrete classes (over all concrete classes) Percentage of final methods

Table 1 continued

Design principle/design heuristic	Reason for being important in APIs	Corresponding measure
Heuristic 2.7 (Riel 1996): "Classes should only exhibit nil or export coupling with other classes, that is, a class should only use operations in the public interface of another class or have nothing to do with that class"	Loose coupling is known to reduce effort and the probability of introducing faults when performing adaptive maintenance and as such it is extremely important to minimize coupling in APIs which by definition undergo adaptive maintenance frequently. Moreover, API designers should strive to minimize accessibility to allow modules to be used, understood, built, tested, and debugged independently (Bloch 2006). According to (Tulach 2008), "*Do Not Expose More Than You Want*", in order to allow future evolution of the API. A direct consequence of minimized accessibility (which maximizes information hiding) is reduced system coupling.	Conformance to this heuristic is expected to be reflected by lower coupling metric values. *Metric*: MPC Message passing coupling refers to the number of messages (method invocations) passing from one class to all those with which it is connected (Li and Henry 1993). Among the import coupling metrics that consider method–method interactions, MPC evaluates coupling employing the total number of method invocations, while the others measure the number of distinct methods invoked. This coupling metric captures both the frequency of connections between classes, as well as the type of dependencies (Briand et al. 1999). The metric value for a class is the sum of MPC for all methods while the system's metric value is calculated as the average over all classes.
Principle of least privilege Modules should only be given access to the resources and information which are necessary for their purpose. related to *Information hiding, encapsulation* (Parnas 1972) and also related to *Design by contract* (Meyer 2000) Design by contract is a programming methodology that guarantees robust software by declaring formal and verifiable specifications for each component. Field invariance conforms to the principle of least privilege and can greatly aid in ensuring that objects are in a consistent state.	API designs should minimize mutability by making objects immutable (Tulach 2008). Classes and members should be immutable offering the advantages of simplicity, testability, thread-safety and reusability (Bloch 2008). Immutable classes have the inherent property of being safe for simultaneous access from multiple threads. Moreover, when using immutable classes it is easier to build a mental model of the system since there is no associated effort to think about how instances will behave at runtime (immutable fields can only be in a single state). The first step to design immutable classes is to ensure that all fields in the class are declared final (Tulach 2008).	Since immutability can be easily enforced by declaring fields as final, the corresponding metric quantifies the extent by which final is employed in the declaration of fields: *Metric*: Percentage of final fields

Table 2 *t*-Test statistic results

Metric	Sig. (2-tailed)	Mean difference*
Abstraction level	0.000	−0.163
Perc. final methods	0.628	
Perc. final classes	0.293	
Perc. final fields	0.005	−0.148
MPC	0.005	10.677

* Sample mean for App group–sample mean for API group

Even if comparison by statistical analysis was flawless and not subject to any threat to validity, the results indicate that safe and clear conclusions cannot be drawn. Moreover, even if metrics are normalized over some global system measure (e.g. averaged over all classes), they neglect the fact that some of the systems are significantly larger in terms of functionality compared to others. The system's size and relevant complexity obviously affects all design decisions in unknown ways and cannot be neglected when comparing software systems of unequal dimensions. Comparing and especially benchmarking software systems neglecting their size is similar to comparing the structural quality of a skyscraper and a one-story building simply by measuring the steel fibers per cubic meter of concrete.

4 Data Envelopment Analysis

Data Envelopment Analysis, initially proposed by Charnes, Cooper and Rhodes (Charnes et al. 1978) is a non-parametric approach that can be used to measure the performance of a number of Decision Making Units. Performance evaluation is achieved by constructing a discrete piecewise frontier over the data and by calculating a maximal performance measure for each DMU in relation to all the other DMUs. Let us consider the simplified case of DMUs having a single input and output. Their efficiency can be calculated as the ratio of output over input. In Fig. 1, a number of sampled DMUs are represented by the

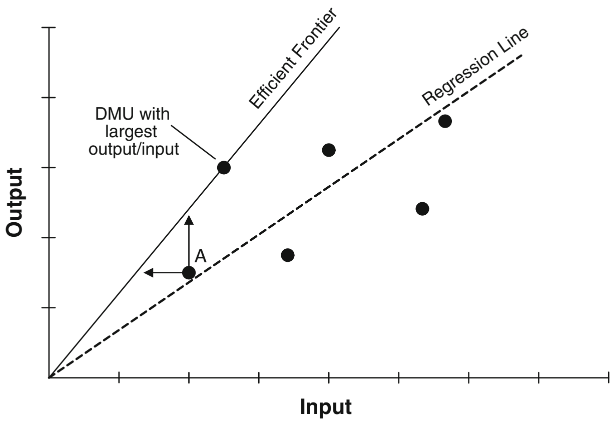

Fig. 1 Efficient frontier vs. regression line [adapted from Cooper et al. (2007)]

corresponding dots, and the slope of the line from the origin through each dot represents the efficiency of each DMU. The line corresponding to the most efficient DMU defines the so-called *efficient frontier*. This frontier envelops all other DMUs (Cooper et al. 2007) whose efficiency is obtained by their distance to this line. On the contrary, a conventional statistical regression approach identifies a central tendency for the selected DMUs (Fig. 1). DMUs in this case are compared to this theoretical baseline rather than to the best performing peer. This signifies the most important difference between DEA and regression approaches.

Figure 1 also illustrates the second benefit from employing DEA. An inefficient DMU can be made efficient in several ways (for a multi input–output model). For example, the DMU represented by dot A can be made efficient either by increasing its output or by decreasing its input. Both actions can move the DMU closer to the efficient frontier.

The relative efficiency of any DMU is obtained as the ratio of a weighted sum of *m* outputs to a weighted sum of *n* inputs. Weights are selected in a manner that the efficiency measure of each DMU is maximized, subject to the constraint that no DMU can have a relative efficiency score greater than unity (Cooper et al. 2007): This can be formulated for a given DMU (e.g. DMU 1) as:

$$\begin{aligned} &\max_{\mathbf{u},\mathbf{v}} \mathbf{u}^T \cdot \mathbf{y}_1 / \mathbf{v}^T \cdot \mathbf{x}_1 \\ \text{subject to}\quad &\mathbf{u}^T \cdot \mathbf{y}_i / \mathbf{v}^T \cdot \mathbf{x}_i \leq 1 \quad \forall i = 1, 2, \ldots, k \\ &\mathbf{u}, \mathbf{v} \geq 0 \end{aligned}$$

where $\boldsymbol{u}$ is an m $\times$ 1 vector of output weights, $\boldsymbol{v}$ is an n $\times$ 1 vector of input weights, $\boldsymbol{y_i}$ is an m $\times$ 1 vector of output values of DMU i, $\boldsymbol{x_i}$ is an n $\times$ 1 vector of input values of DMU i, and k the number of DMUs.

Usually the above fractional problem is transformed into a linear programming problem (*multiplier* form) by equating the denominator of the efficiency ratio of the DMU under study to unity. Then, by using the equivalent dual model, the DEA problem takes the following *envelopment* form:

$$\begin{aligned} &\min_{\theta,\lambda} \theta \\ \text{subject to}\quad &-\mathbf{y}_1 + \mathbf{Y} \cdot \boldsymbol{\lambda} \geq \mathbf{0} \\ &\theta \cdot \mathbf{x}_1 - \mathbf{X} \cdot \boldsymbol{\lambda} \geq \mathbf{0} \\ &\boldsymbol{\lambda} \geq \mathbf{0} \end{aligned}$$

where θ is the efficiency score of DMU 1 ($0 < \theta \leq 1$), $\boldsymbol{\lambda}$ is a k $\times$ 1 vector of constants, $\mathbf{Y}$ is the m $\times$ k output matrix, $\mathbf{X}$ is the n $\times$ k input matrix.

The above model is the so-called CCR model (CCR stands for the initials of the authors who proposed the model), which assumes constant returns to scale. A production function is said to exhibit constant returns to scale (CRS) if a proportionate increase in all inputs results in the same proportionate increase in output (Coelli et al. 2005). The assumption of constant returns to scale is only appropriate when all DMUs operate at an optimal scale and there is a proportional relationship between inputs and outputs. If this is not a valid assumption, as occurs in our study, the CCR model should be extended to account for variable returns to scale. The most representative model, which has been proposed so far for variable returns to scale, is the BCC model (Banker et al. 1984) (BCC also stands for the initials of the authors). This model has been employed in our study.

In summary, the main advantages of DEA over other approaches are:

- DEA can handle multiple inputs and outputs.
- Inputs and outputs can have varying measurement scales.
- Based on projections of the inefficient DMUs onto the efficient frontier, estimates for improvements in inputs and/or outputs can be produced.
- DMUs are directly compared against a peer or a combination of peers and not against a theoretical baseline, making the approach appropriate for benchmarking.

Application of DEA in order to evaluate and benchmark a purely technological aspect of software development differs from the conventional use of DEA in an economic context regarding the meaning of inputs. Since according to the proposed model the inputs to the software design process refer to the behavior and state that the designer has to implement in the system, they differ from conventional inputs in an economic context. Inputs, such as investments, number of employees or salaries are subject to optimization, whereas the number of methods and attributes that should exist in a software design are not negotiable (considering that a flawless object-oriented analysis has preceded the design phase).

To illustrate this difference between inputs in an economic context and inputs to a software design process, consider the simplified examples with a single input and single output, one from each domain, shown in Fig. 2.

As already mentioned, in an economic context, the overall goal is to maximize the outputs and at the same time to minimize inputs, i.e. the criterion for benchmarking a simplified company as in Fig. 2 can be stated as Performance $= \frac{\text{output}}{\text{input}}$. In other words, between two companies with the same output, one would consider as better the smaller one, e.g. the one that achieves this output with the smaller investment (smaller input).

On the other hand, for a software design process, the input is fixed and not subject to optimization. However, to understand the difference in the technological context, consider the following example case: let us assume that two software designs have the same abstraction level (same value for the output). For these systems, we usually regard as better achievement the larger one, in the sense that it achieves the same abstraction level despite the fact that it is larger in size, in terms of operations or any other size measure. To state this mathematically (although inputs are fixed), the criterion for benchmarking software designs can be stated as Performance = input $\times$ output.

Therefore, we model the inputs to DEA for the software products that have been analyzed as $\frac{1}{\text{input metric}}$, for each one of the two selected input metrics.

A reasonable concern regarding the choice of inputs could be that methods and attributes are not the actual inputs of the software design process. Indeed, when the software analysis and design process is viewed as a whole, a set of given requirements can be implemented in several ways, corresponding to varying sets of methods and attributes. In this context, attribute and method selection and allocation is obviously negotiable.

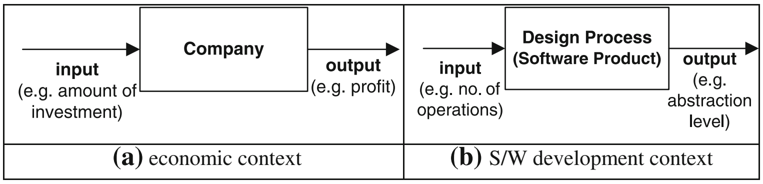

Fig. 2 Simplified examples comparing economic and software development contexts

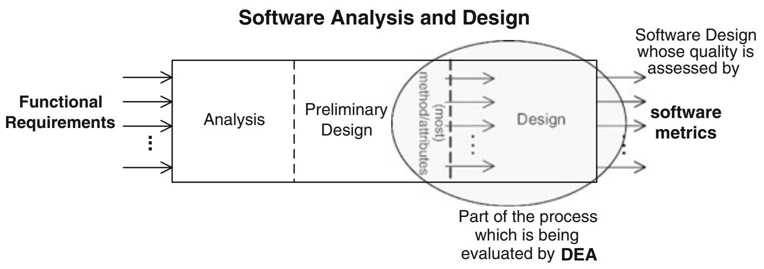
Software Analysis and Design
Functional Requirements
Analysis
Preliminary Design
(most) method/attributes
Design
Software Design whose quality is assessed by
software metrics
Part of the process which is being evaluated by DEA

Fig. 3 Relaxed view of the software design process forming the context of DEA

However, in the context of Data Envelopment Analysis, it would not make sense to suggest an artificial reduction (or increase) in the number of attributes and methods, just to improve the efficiency of a software project in terms of some of its metrics. The ideal would be to employ as input to DEA measures of functionality that are provided by the system (such as function points). However, in the lack of well-established metrics of functionality that can be extracted from source code, we opted for the alternative of employing the number of methods and attributes as an indicator of the requested functionality and amount of state information. The analogy can be found in an object-oriented analysis and design methodology (e.g. the ICONIX methodology), in which most domain-related attributes and operations have been extracted with the completion of the preliminary design phase. In this context, it can be assumed (neglecting the iterative nature of current processes) that attributes and methods act as input to the design process where methods are allocated to classes, class relationships are finalized, design principles are followed and design patterns are applied if appropriate. This consideration, which defines the environment in which DEA is applied, is illustrated in Fig. 3.

For the selected outputs, the designer's goal should be to maximize the abstraction level and the percentage of final classes, methods and fields in the system. However, for MPC, the goal is to minimize its value as it expresses coupling. To be consistent with the other outputs and be able to employ the BCC approach, we model the corresponding output as $1/MPC$, i.e. the values fed to DEA analysis are the inverse of MPC metric value.

5 Inputs and data

5.1 Input variables

As already mentioned, there are a number of approaches in the literature that treat the design of an object-oriented system as a multi-objective optimization problem in which the goal is to reassign methods and attributes to classes (either from scratch or by employing stepwise modifications in the form of refactorings) in order to optimize the value of selected metrics, typically coupling and cohesion (O'Keeffe and O'Cinneide 2006; Seng et al. 2006; Bowman et al. 2007). The common denominator of these approaches is that the "independent" variables of a software design are its behavior, expressed by the methods that provide the functionality and its state, expressed by the attributes holding the system's information. Since the goal is to express the functionality that is required from the system under development, we count only concrete methods that have an implementation. Therefore, we select as inputs to DEA the following two measures:

Number of Concrete Methods: Total number of implemented methods in a class, regardless of access specifier. The metric value for a system is calculated as the sum over all classes.

Number of Attributes (NOA): Total number of attributes defined in a class, regardless of access specifier (Lorenz and Kidd 1994). The metric value for a system is calculated as the sum over all classes.

Since these two measures are actually extracted from the source code (i.e. after the design has been completed), it could be argued that they are also output metrics, in the sense that the software designer assigns responsibilities and attributes to the classes of the system. However, as already mentioned, in the absence of reliable tools for counting other measures of functionality and considering that for the same methods and attributes numerous alternatives for a software design exist, we can regard them as indicators of the functionality that is requested to be implemented and given supposedly as input to the design team.

5.2 Data

To achieve efficiency discrimination among DMUs, DEA requires that the number of DMUs is significantly larger than the number of inputs and outputs. For n inputs and m outputs, a rule of thumb (Cooper et al. 2007) suggests that the number of DMUs should be $\#DMUs \geq \max\{n \times m, 3 \times (n + m)\}$. From the previous analysis, we have 2 inputs and 5 outputs. Therefore, at least 21 DMUs should be included in the analysis. We have analyzed 44 software projects.

The selection of the projects that have been chosen as DMUs was based on the following criteria:

- The projects should be open-source in order to allow the calculation of the corresponding metric values from source code.
- Projects should have diverse size characteristics to evaluate DEA's ability to handle DMUs that have varying mix of inputs/outputs.
- The pool of projects should contain mature projects (e.g. APIs that have been constantly evolving for a number of years) that are expected to have a relatively good performance, as well as immature and possibly poorly structured projects that are expected to have low efficiency. The maturity of each project is estimated considering the date when the project was registered in an open-source repository and the date on which the latest file was committed. A long period between the two dates implies, in most of the cases, a mature and active project.
- Projects should be from several domains to limit the threats to external validity.
- Projects should be written in the same programming language (i.e. Java) to minimize any effect of the programming language on the calculation of metrics.

The projects that have been included in the analysis, along with a brief description, are listed in Table 3.

An overview of the data for the above referenced projects that have been used as inputs and outputs for DEA along with two size measures (LOC—Lines of Code and NOC—Number of Classes) is given in Table 4. LOC and NOC are provided only for reference and have not been used as inputs to the DEA analysis since they result from the design process and cannot be considered as inputs to it.

It should be noted that correlated inputs and outputs do not distort the calculated efficiency scores. According to the developers of DEA (Charnes et al. 1995), high correlation coefficients do not prevent us from running a DEA model because of the non-parametric

Table 3 Set of projects (DMUs) under study

Libraries			Applications		
1	javax.sql	API for server side data source access and processing	1	JHotDraw 5.1.4	GUI framework for technical and structured graphics
2	javax.sound	Java low-level API for creating, modifying, and controlling the input and output of sound media, including both audio and MIDI	2	Violet 0.16a	Cross-platform, easy to use UML editor
3	javax.xml	Defines core XML constants and functionality from the XML specifications	3	Jeppers	Web-based spreadsheet editor
4	java.awt	Collection of original Java classes for creating user interfaces and for painting graphics and images	4	JMol 9.0	Molecular viewer for three-dimensional chemical structures
5	java.io	Resources for system input and output through data streams, serialization and the file system	5	JEdit 4.0	Programmer's text editor that can be configured as an IDE
6	Netbeans 5.0 Debugger Core	Definitions of common structures for integration of debugger implementations into the NetBeans IDE	6	GanttProject 2.0.9	Project scheduling application featuring Gantt charts, resource management, calendaring, import/export
7	Netbeans 5.0 UI Utilities	Utility classes pertaining to the visual appearance of the IDE	7	EJE—Everyone's Java Editor 2.7	Simple light-weight Java editor
8	JFreeChart 0.7	Java chart library that allows developers to display charts in their applications	8	Compiere 2.4.4	ERP solution for distribution, retail, manufacturing and service industries that automates accounting, supply chain, inventory and sales orders
9	Algorithm Study 0.2.0	Provides implementations of algorithms (sorting, searching, etc.) and data structures (lists, trees, etc.)	9	Franklin Math 0.11	A computer algebra system (CAS) that supports both numeric and symbolic arithmetic and other computations
10	JDOM 1.1	Provides a Java-based solution for accessing, manipulating, and outputting XML data from Java code	10	StatSVN 0.5.0	Metrics-analysis tool for charting software evolution through analysis of Subversion source repositories
11	org.eclipse. core.filesystem	Provides an interface for interacting with a file system	11	JFigure 1.0.8	Application for drawing dynamic mathematics features (as geometric figures, dynamic algebra) and for creating dynamic animations using geometrics tools
12	org.eclipse.ui.views	Application programming interfaces for interaction with and extension of the eclipse platform user interface	12	Jns 1.7	Java version of the ns-2 network simulator originally from Berkeley. It allows developers of routing and other network protocols to simulate their protocols under various conditions.

Table 3 continued

	Libraries			Applications	
13	Jasperreports 3.5.0	Java reporting library that delivers sophisticated print or web reports	13	JSpider 0.5.0	A highly configurable and customizable web spider engine
14	BCEL 5.2	Byte code engineering library that is intended to give users a possibility to analyze, create, and manipulate (binary) Java class files. Part of the Apache Jakarta project	14	JSignpdf 0.8.0	Application which adds digital signatures to PDF documents
15	Sax 2r3	Simple API for XML	15	JFlex 1.4.3	Lexical analyzer generator for Java
16	Apache Commons Collections 3.2	Collection of open-source reusable Java components from the Apache/Jakarta community	16	JDepend 2.9	A Java package dependency analyzer that generates design quality metrics
17	Trove 2.1.0	High performance collections for Java objects and primitive types	17	BlueJ 2.5.1	Java IDE specifically designed to learn and teach object-oriented programming and Java
18	Mango	Java library consisting of a number of iterators, algorithms and functions, loosely inspired by the C++ STL	18	PMD 4.1	Java source code analyzer that looks for potential problems
19	JGAP 3.01	Genetic algorithms and genetic programming component provided as a Java framework	19	FreeCol 0.7.2	Turn-based strategy game similar to Civilization
20	ASM 2.0	All purpose Java bytecode manipulation and analysis framework	20	Robocode 1.5.1	Programming game where the goal is to develop a robot battle tank to battle against other tanks
21	Guava r03	Google's core Java libraries	21	BeautyJ 1.1	Source code transformation tool for Java source files that generates a clean, normalized representation of the code
22	JMeasurement 0.70.129	Java API for monitoring runtime and usage of user defined points in java production code	22	JAllInOne 0.9.10	ERP/CRM Java application having a Swing front-end

Table 4 Project inputs and outputs

	Project	LOC	NOC	Inputs		Outputs				
				# Concrete methods	# Fields	% (Abstract classes + interfaces)	% Final classes	% Final methods	MPC	% Final fields
Libraries										
1	javax.sql	14643	47	313	140	0.574	0.000	0.000	13.457	0.136
2	javax.sound	11905	70	262	164	0.471	0.000	0.094	10.852	0.518
3	javax.xml	35323	274	719	438	0.613	0.132	0.017	6.197	0.658
4	java.awt	147557	482	4578	3258	0.349	0.197	0.048	28.278	0.573
5	java.io	26128	109	974	420	0.202	0.057	0.058	28.906	0.436
6	Netbeans 5 Debugger Core	3921	34	193	98	0.471	0.333	0.023	27.478	0.388
7	Netbeans 5.0 UI Utilities	9692	67	425	244	0.194	0.167	0.031	51.769	0.340
8	JFreeChart 0.7	19098	110	738	390	0.245	0.000	0.000	30.178	0.256
9	Algorithm Study 0.2.0	2578	30	115	99	0.167	0.000	0.000	4.500	0.152
10	JDOM 1.1	18081	73	720	255	0.123	0.125	0.015	33.565	0.427
11	org.eclipse.core.filesystem	3488	22	157	92	0.363	0.000	0.052	7.682	0.500
12	org.eclipse.ui.views	5143	37	220	108	0.324	0.040	0.000	6.722	0.157
13	Jasperreports 3.5.0	197741	1330	10649	6104	0.239	0.008	0.009	24.751	0.471
14	BCEL 5.2	38423	357	2255	982	0.199	0.203	0.146	20.370	0.560
15	Sax 2r3	8781	41	275	71	0.390	0.160	0.000	11.711	0.141
16	Apache Commons Coll. 3.2	24635	412	3087	763	0.147	0.149	0.005	17.485	0.498
17	Trove 2.1.0	3250	54	280	71	0.302	0.170	0.078	14.741	0.465
18	Mango	1837	81	254	53	0.083	0.000	0.000	6.038	0.415
19	JGAP 3.01	15061	242	1306	718	0.314	0.008	0.004	15.504	0.410
20	ASM 2.0	14134	146	906	705	0.130	0.295	0.100	20.589	0.274
21	Guava r03	22735	465	2838	723	0.224	0.251	0.018	13.204	0.743
22	JMeasurement 0.70.129	9558	104	660	381	0.115	0.231	0.340	27.173	0.412

Table 4 continued

	Project	LOC	NOC	Inputs		Outputs				
				# Concrete methods	# Fields	% (Abstract classes + interfaces)	% Final classes	% Final methods	MPC	% Final fields
Applications										
1	JHotDraw 5.1.4	13030	158	1000	334	0.196	0.024	0.002	19.013	0.225
2	Violet 0.16a	8272	74	364	261	0.162	0.000	0.000	26.687	0.284
3	Jeppers	2909	21	113	93	0.000	0.000	0.000	31.571	0.011
4	JMol 9.0	41969	316	1956	1824	0.057	0.020	0.000	39.085	0.254
5	JEdit 4.0	89138	558	3496	2244	0.082	0.031	0.178	48.975	0.176
6	GanttProject 2.0.9	59323	733	3875	2565	0.216	0.007	0.001	28.695	0.363
7	EJE 2.7	9371	97	438	450	0.093	0.000	0.007	38.230	0.138
8	Compiere 2.4.4	200221	737	7749	5317	0.069	0.165	0.007	45.348	0.275
9	Franklin Math 0.11	9096	72	412	293	0.069	0.119	0.086	17.868	0.147
10	StatSVN 0.5.0	6470	41	324	202	0.049	0.308	0.009	40.811	0.559
11	JFigure 1.0.8	126506	883	5855	3311	0.049	0.088	0.342	32.569	0.310
12	jns 1.7	5874	61	248	205	0.197	0.000	0.000	11.246	0.195
13	JSpider 0.5.0	10986	251	884	547	0.311	0.000	0.003	9.502	0.346
14	JSignpdf 0.8.0	6276	30	337	335	0.033	0.034	0.003	39.643	0.412
15	JFlex 1.4.3	15163	57	447	491	0.105	0.373	0.035	28.630	0.242
16	JDepend 2.9	2379	29	283	86	0.103	0.000	0.000	37.483	0.198
17	BlueJ 2.5.1	76352	693	4487	2913	0.132	0.130	0.063	28.568	0.270
18	PMD 4.1	42626	595	3298	1667	0.121	0.020	0.110	17.889	0.208
19	FreeCol 0.7.2	60754	486	3523	3517	0.104	0.347	0.017	47.922	0.737
20	Robocode 1.5.1	26754	265	2401	1413	0.049	0.045	0.027	36.479	0.167
21	BeautyJ 1.1	22514	242	1746	688	0.459	0.017	0.261	24.694	0.135
22	JAllInOne 0.9.10	132795	1239	7241	8044	0.008	0.000	0.171	5.154	0.031

Metric values have been extracted by analyzing all projects with a custom-developed Eclipse plugin. LOC and MPC have been extracted with Borland Together 2006 for Eclipse nature of DEA, which mitigates this effect. Moreover, DEA remains unaffected when outputs are scaled by inputs as in the presented model where the number of final methods and fields (outputs) is averaged over the total number of methods and fields (inputs), respectively. According to Dyson et al. (2001), if scale is thought of as the physical size of a unit (as it happens for the aforementioned software metrics), then scaling by input is appropriate. Moreover, a major pitfall would occur if ratios (as the other three outputs) had been mixed with volume measures. This would lead to improper comparisons (we observed this problem initially in our experiments) and is avoided by averaging the two outputs over the corresponding inputs.

6 Results and discussion

6.1 Normal DEA model

Data Envelopment Analysis has been performed on the selected projects/metrics employing the DEA-Solver v6.0 software by SAITECH Inc. The results are shown in Table 5 (rows in italics correspond to libraries). The 3rd column corresponds to the overall efficiency score calculated for each project. Projects are ranked (1st column) according to this score.

As it can be observed, 15 projects are considered as fully efficient (i.e. have an efficiency score equal to one and no shortages or excesses in outputs), with nine of them belonging to the group of libraries. Moreover, the average efficiency score for libraries is 0.845, whereas the average efficiency score for applications is 0.629, confirming our initial expectation about the superiority of libraries, in the context of the selected metrics. The difference between the means of the two groups is according to an independent sample *t*-test statistically significant ($p = 0.005$).

To provide an overview on what could be improved in each project, the rest of the columns in Table 5 show the differences between the actual and the expected data when each inefficient software design is projected onto the efficient frontier. In other words, these columns indicate the required changes on the selected metrics in order to make the efficiency score of an inefficient project equal to one. Differences for the input metrics are not shown, since the inputs to the design process represented by number of operations and attributes, are considered in the proposed model as not negotiable, as already mentioned.

According to the theory of DEA, the efficient projects are not in need of any improvement, within the context of the examined metrics, since they are placed on the efficient frontier. Having a look at the marginally inefficient projects [with an efficiency score between 0.9 and 1.0 (von Mayrhauser et al. 2000)], i.e. projects javax.sql, javax.sound and Mango, they show a tremendous hysteresis in the number of final classes, since according to the data they all have zero classes declared as final providing room for subclassing (javax.sql and Mango have also a very small percentage of final methods). Thus, an increase in the corresponding metrics is suggested vividly by the model.

Results should be viewed in light of the assumptions that have been made earlier. For example, if the projections indicate that one of the outputs should be improved (e.g. the percentage of abstract classes and interfaces), this does not necessarily mean that the system suffers from a serious design problem. But in the context of benchmarking, another project with a similar mix of inputs/outputs and a better percentage of abstract classes and interfaces is viewed as more efficient. As already mentioned, in the case where two projects have the same or similar outputs (i.e. the same quality level according to the

Table 5 Efficiency scores and projections onto the efficient frontier

Rank	Project	Efficiency	Difference between actual and projected value (%)				
			% (Abstract classes + interfaces)	% Final classes	% Final methods	MPC	% Final fields
1	*JMeasurement 0.70.129*	*1.000*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
1	*Guava r03*	*1.000*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
1	*ASM 2.0*	*1.000*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
1	*Jasperreports 3.5.0*	*1.000*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
1	*Algorithm Study 0.2.0*	*1.000*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
1	*Netbeans 5.0 Debugger Core*	*1.000*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
1	JFigure 1.0.8	1.000	0.00	0.00	0.00	0.00	0.00
1	*BCEL 5.2*	*1.000*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
1	Compiere 2.4.4	1.000	0.00	0.00	0.00	0.00	0.00
1	*java.awt*	*1.000*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
1	*javax.xml*	*1.000*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
1	JFlex 1.4.3	1.000	0.00	0.00	0.00	0.00	0.00
1	JAllInOne 0.9.10	1.000	0.00	0.00	0.00	0.00	0.00
1	BeautyJ 1.1	1.000	0.00	0.00	0.00	0.00	0.00
1	FreeCol 0.7.2	1.000	0.00	0.00	0.00	0.00	0.00
16	*javax.sql*	*0.937*	*6.73*	*999.90*	*999.90*	*−53.95*	*384.50*
17	*javax.sound*	*0.930*	*7.56*	*999.90*	*7.56*	*−26.88*	*7.56*
18	*Mango*	*0.901*	*427.55*	*999.90*	*999.90*	*−9.90*	*10.99*
19	StatSVN 0.5.0	0.888	141.71	12.66	131.42	−11.24	12.66
20	*org.eclipse.core.filesystem*	*0.856*	*42.27*	*999.90*	*16.78*	*−14.37*	*16.78*
21	*Trove 2.1.0*	*0.792*	*26.24*	*26.24*	*26.24*	*−27.07*	*26.24*
22	*org.eclipse.ui.views*	*0.790*	*26.56*	*82.00*	*999.90*	*−20.98*	*172.34*
23	*Sax 2r3*	*0.751*	*33.08*	*33.08*	*999.90*	*−24.86*	*282.77*
24	*Apache Commons Coll. 3.2*	*0.723*	*45.84*	*48.88*	*398.62*	*−27.73*	*38.37*
25	*JGAP 3.01*	*0.672*	*48.79*	*999.90*	*760.20*	*−32.79*	*48.79*
26	*java.io*	*0.669*	*49.38*	*275.90*	*49.38*	*−61.40*	*49.38*
27	GanttProject 2.0.9	0.667	50.02	999.90	999.90	−33.34	50.02
28	BlueJ 2.5.1	0.662	50.99	50.99	50.99	−33.77	90.76
29	JSpider 0.5.0	0.640	56.63	999.90	999.90	−36.02	56.29
30	PMD 4.1	0.624	60.16	382.05	60.16	−37.56	60.16
31	JEdit 4.0	0.603	65.84	194.05	65.84	−39.70	80.90
32	*Netbeans 5.0 UI Utilities*	*0.589*	*69.87*	*69.87*	*69.87*	*−66.84*	*69.87*
33	Franklin Math 0.11	0.586	72.57	70.66	70.66	−41.40	70.66
34	*JDOM 1.1*	*0.582*	*79.34*	*100.37*	*71.76*	*−60.12*	*71.76*
35	JSignpdf 0.8.0	0.554	572.01	627.90	508.52	−66.69	80.37
36	jns 1.7	0.480	108.38	999.90	999.90	−52.01	121.25
37	*JFreechart 0.7*	*0.405*	*147.05*	*999.90*	*999.90*	*−78.77*	*147.52*
38	Violet 0.16a	0.403	148.23	999.90	999.90	−67.42	148.23
39	JMol 9.0	0.401	187.80	999.90	999.90	−59.87	149.18

Table 5 continued

Rank	Project	Efficiency	Difference between actual and projected value (%)				
			% (Abstract classes + interfaces)	% Final classes	% Final methods	MPC	% Final fields
40	JHotDraw 5.1.4	0.394	153.71	479.40	999.90	−60.59	160.17
41	Robocode 1.5.1	0.315	249.08	343.61	217.62	−68.52	217.62
42	JDepend 2.9	0.278	259.07	999.90	999.90	−75.26	259.07
43	EJE 2.7	0.204	400.46	999.90	389.07	−79.55	389.07
44	Jeppers	0.143	999.90	0.00	0.00	−85.75	999.90

Rows in italics correspond to libraries

selected metrics), the approach ranks as better the one with larger inputs, i.e. the one that is larger in terms of functionality and state variables.

If the model results are interpreted accurately, the differences provide a form of guidelines on what should be improved in each project, when comparing it to the most efficient projects. For example, library JFreeChart is less efficient (score: 0.405) than javax.xml (score: 1). The projection indicates that in the context of this analysis all of its metrics could be improved. What the model captures is the fact that javax.xml, which is a fully efficient project that is most directly comparable to JFreeChart (because they have a similar mix of inputs and outputs), excels in all of the five output metrics and therefore the model expects from JFreeChart significantly improved outputs in all of the examined aspects. As a result, the information to the designers of JFreeChart is to learn from javax.xml and using it as a baseline to attempt to improve the corresponding design properties of JFreeChart. The information that is provided by the model, combined with other objective or subjective sources of information, can help the design team of any project to establish a golden set of projects, whose best practices should be emulated. For example, the overall high efficiency scores for all java libraries combined with the fact that JDK API is considered to be professionally designed (Tulach 2008) strengthens the belief that Java libraries are safe to rely upon and to follow as an example for API design.

As another example, JHotDraw 5.1.4, which is a well-known project widely acknowledged for its proper use of design patterns, is unexpectedly ranked 40th. The projection indicates that in the context of this analysis all of its metrics could be improved. Compared to javax.xml, which is also a project having inputs that are roughly similar, JHotDraw appears to have a significantly lower percentage of abstract classes and interfaces, final classes, final methods and final fields and a larger MPC value (see Table 4). This is also confirmed by the *reference sets* or *peer groups* that are provided by DEA for each inefficient DMU. The reference set consists of those efficient peers that operate closer to a given DMU considering their mix of inputs and outputs. In other words, it provides for the inefficient DMUs the efficient ones with which they are most directly comparable. It is the existence of these efficient peers that forces a DMU to be inefficient. Table 6 provides the reference sets for the inefficient projects. This is another valuable aspect of DEA, since it provides efficient projects to which a given system can be compared in order to gain insight into what and how much can be improved. As von Mayrhauser et al. (2000) suggest, projects which according to subjective analysis are considered successful and according to a production model like DEA are indicated as efficient are the ones that we would like to learn from.

Table 6 Reference sets for the inefficient DMUs

Project	Score	Reference set
javax.sql	0.937	javax.xml
javax.sound	0.930	javax.xml, JMeasurement, BeautyJ
Mango	0.901	javax.xml, AlgorithmStudy, JAllInOne
StatSVN 0.5.0	0.888	FreeCol, JFlex, javax.xml
org.eclipse.core.filesystem	0.856	javax.xml, JMeasurement, JAllInOne
Trove 2.1.0	0.792	javax.xml, JMeasurement, FreeCol, Netbeans 5.0 Debug. Core
org.eclipse.ui.views	0.790	javax.xml, AlgorithmStudy, JAllInOne
Sax 2r3	0.751	javax.xml, Netbeans 5.0 Debug. Core, JFlex
Apache Commons Coll. 3.2	0.723	Guava, Jasperreports, JAllInOne
JGAP 3.01	0.672	java.awt, javax.xml, Guava JAllInOne
java.io	0.669	Guava, javax.xml, JMeasurement
Gantt project 2.0.9	0.667	java.awt, JAllInOne, javax.xml, Guava, FreeCol
BlueJ 2.5.1	0.662	java.awt, FreeCol, JFigure, JAllInOne, Guava
JSpider 0.5.0	0.640	javax.xml, JAllInOne, FreeCol
PMD 4.1	0.624	JFigure, java.awt, JAllInOne, javax.xml, BeautyJ
JEdit 4.0	0.603	JFigure, BeautyJ, java.awt, JAllInOne
Netbeans 5.0 UI Utilities	0.589	FreeCol, Netbeans 5.0 Debug. Core, javax.xml, JMeasurement
Franklin Math 0.11	0.586	JFlex, JAllInOne, JMeasurement, javax.xml
JDOM 1.1	0.582	Guava, JMeasurement
JSignpdf 0.8.0	0.554	Guava
jns 1.7	0.480	javax.xml, AlgorithmStudy, JAllInOne
JFreechart 0.7	0.405	javax.xml, BeautyJ
Violet 0.16a	0.403	Guava, javax.xml
JMol 9.0	0.401	FreeCol, javax.xml, JAllInOne
JHotDraw 5.1.4	0.394	javax.xml, java.awt, JAllInOne
Robocode 1.5.1	0.315	FreeCol, JAllInOne, javax.xml, Guava, JFigure
JDepend 2.9	0.278	Guava, javax.xml
EJE 2.7	0.204	javax.xml, Guava, JMeasurement
Jeppers	0.143	AlgorithmStudy

6.2 System differentiated DEA model

The results concerning the software categories that have been examined might have been affected by the different inherent characteristics of libraries and applications. Recent extensions to basic DEA models attempt to address this issue by the introduction of categorical variables in the analysis and the corresponding modification of the linear programming formulation, forming the so-called system differentiated DEA (SYS-DEA), which allows cross-system comparisons (Cooper et al. 2007). System differentiated DEA is appropriate when the operating environment of the examined DMUs exhibits systematic differences (Yang 2009). In this way, it is not only possible to evaluate the efficiency of each DMU but also to compare the two or more categories by observing the efficiency of DMUs in each system. The results of system differentiated DEA are given in Table 7.

Table 7 Efficiency scores and projections onto the efficient frontier according to system differentiated DEA

Rank	Project	Efficiency	Difference between actual and projected value (%)				
			% (Abstract classes + interfaces)	% Final classes	% Final methods	MPC	% Final fields
1	*JMeasurement 0.70.129*	*1.000*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
1	*Guava r03*	*1.000*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
1	*ASM 2.0*	*1.000*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
1	*Jasperreports 3.5.0*	*1.000*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
1	*Algorithm Study 0.2.0*	*1.000*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
1	*Netbeans 5.0 Debugger Core*	*1.000*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
1	JFigure 1.0.8	1.000	0.00	0.00	0.00	0.00	0.00
1	*BCEL 5.2*	*1.000*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
1	Compiere 2.4.4	1.000	0.00	0.00	0.00	0.00	0.00
1	*java.awt*	*1.000*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
1	*javax.xml*	*1.000*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
1	JFlex 1.4.3	1.000	0.00	0.00	0.00	0.00	0.00
1	JAllInOne 0.9.10	1.000	0.00	0.00	0.00	0.00	0.00
1	BeautyJ 1.1	1.000	0.00	0.00	0.00	0.00	0.00
1	FreeCol 0.7.2	1.000	0.00	0.00	0.00	0.00	0.00
16	BlueJ 2.5.1	0.991	0.86	0.86	0.86	−0.85	35.73
17	*javax.sound*	*0.964*	*3.71*	*999.90*	*3.71*	*−29.28*	*15.05*
18	*javax.sql*	*0.937*	*6.73*	*999.90*	*999.90*	*−53.95*	*384.50*
19	*Mango*	*0.907*	*451.22*	*999.90*	*999.90*	*−9.29*	*15.94*
20	StatSVN 0.5.0	0.894	109.35	11.88	165.76	−10.62	11.88
21	*org.eclipse.core.filesystem*	*0.867*	*38.45*	*999.90*	*15.28*	*−13.25*	*15.28*
22	*Trove 2.1.0*	*0.800*	*24.98*	*24.98*	*24.98*	*−27.13*	*24.98*
23	*Apache Commons Coll. 3.2*	*0.796*	*53.51*	*50.52*	*240.17*	*−20.40*	*43.19*
24	*org.eclipse.ui.views*	*0.793*	*29.60*	*87.59*	*999.90*	*−20.72*	*178.90*
25	JSpider 0.5.0	0.777	71.47	999.90	522.09	−22.27	78.71
26	*Sax 2r3*	*0.759*	*42.83*	*31.73*	*999.90*	*−24.09*	*291.95*
27	GanttProject 2.0.9	0.707	41.37	999.90	999.90	−29.26	42.93
28	PMD 4.1	0.701	42.58	419.20	42.58	−29.86	42.58
29	*JGAP 3.01*	*0.675*	*48.22*	*999.90*	*668.13*	*−32.53*	*48.22*
30	*java.io*	*0.669*	*49.38*	*275.90*	*49.38*	*−61.40*	*49.38*
31	JEdit 4.0	0.659	51.83	108.32	51.83	−37.29	70.89
32	*Netbeans 5.0 UI Utilities*	*0.620*	*61.39*	*61.39*	*61.39*	*−62.60*	*61.39*
33	Franklin Math 0.11	0.593	387.29	68.74	68.74	−40.74	233.82
34	*JDOM 1.1*	*0.582*	*79.34*	*100.37*	*71.76*	*−60.12*	*71.76*
35	JSignpdf 0.8.0	0.554	572.01	627.90	508.52	−66.69	80.37
36	jns 1.7	0.490	136.35	999.90	999.90	−51.02	150.91
37	JMol 9.0	0.440	127.13	999.90	999.90	−55.97	127.13
38	JHotDraw 5.1.4	0.411	143.08	542.97	895.46	−58.86	195.20

Table 7 continued

Rank	Project	Efficiency	Difference between actual and projected value (%)				
			% (Abstract classes + interfaces)	% Final classes	% Final methods	MPC	% Final fields
39	*JFreechart 0.7*	*0.406*	*146.51*	*999.90*	*999.90*	*−78.96*	*155.43*
40	Violet 0.16a	0.403	148.23	999.90	999.90	−67.42	148.23
41	Robocode 1.5.1	0.361	176.68	323.00	176.68	−63.86	176.68
42	JDepend 2.9	0.278	259.07	999.90	999.90	−75.26	259.07
43	EJE 2.7	0.204	400.46	999.90	389.07	−79.55	389.07
44	Jeppers	0.143	999.90	0.00	0.00	−85.75	999.90

Rows in italic correspond to libraries

Table 8 Comparison between normal and system differentiated DEA

	Normal DEA model		System differentiated DEA model	
Number of efficient DMUs	15		15	
	Libraries	Applications	Libraries	Applications
	9	6	9	6
Average efficiency score	0.737		0.759	
	Libraries	Applications	Libraries	Applications
	0.845	0.629	0.853	0.664

Table 8 presents a summary of the comparison between the results of normal DEA and system differentiated DEA. System differentiated analysis yields a slightly higher average efficiency score for all projects and the same number of efficient projects, while the relative ranking order of the projects presents only minor differences. These observations are in agreement with the remarks by Yang (2009) which applied system differentiated DEA to account for differences between several geographical areas in the assessment of Canadian bank branches' performance.

According to the system differentiated DEA model, the average efficiency score of applications is 0.664, whereas the average score of libraries is 0.853, confirming the conclusions derived so far. To test statistically the difference between the two groups in terms of efficiency and to assess whether differences occur by chance or are statistically significant, an independent sample *t*-test may be used, since the distribution of the efficiency scores has been tested for normality. The resulting *p*-value is equal to 0.013. As a result, at the $\alpha = 0.05$ level of significance, there is enough evidence to conclude that there is a difference in the efficiency scores between the two types of software.

7 Threats to validity

7.1 Threats to internal validity

As threats to internal validity, we consider those factors that may cause interferences regarding the relationships that we are trying to investigate (Wohlin et al. 2000). There are two threats related to internal validity and essentially concern incorrect model specification. First of all, important inputs and most probably outputs might have been ignored in the analysis. Since a limited number of design principles have been analyzed with respect to API design, it cannot be claimed that all aspects of the design quality of the examined systems have been accurately captured by the selected metrics. Second, DEA does not impose weights on any of the outputs, treating all output metrics as equally important. This is obviously questionable for a software system since certain design decisions might put emphasis on some aspects of the design, neglecting others. However, as already mentioned, the proposed analysis does not aim to provide an indisputable ranked list of software projects according to their efficiency but to illustrate how DEA can be used for comparing projects when multiple criteria for analysis are in hand. Finally, software systems from different domains might have fundamental differences in inherent complexity and therefore in the way that functionality and state affect design and implementation (the underlying production function that the model explores). To confront this threat, future research could investigate the use of categorical input variables to allow for further differences, such as the software domain, to obtain more refined evaluations and insights.

7.2 Threats to external validity

As threats to external validity, we consider those factors that limit the possibility to generalize the DEA findings beyond the immediate study to other settings. Obviously, a different set of projects would lead to a different ranked list for APIs and applications. This kind of threat is always valid in an empirical study when the number of systems is limited and the criticism is related to possible differences between the projects that have been selected for analysis and other kinds of projects.

8 Related work

In the literature of software engineering, there is a consensus among researchers that metric values should be combined in order to extract valuable information and several approaches have been proposed in this direction. In the work by Yamashita et al. (2009) concept mapping is proposed as a means to assess software maintainability by incorporating multiple metrics. Concept mapping, whose origin lies in social research, aims at structuring the knowledge for a domain by specifying pertinent entities and the relations between them. The final outcome is a conceptual representation of the elements under analysis where logical groups of concepts form clusters. The approach is in line with Arisholm and Sjøberg's observation (Arisholm and Sjøberg 2004) that metrics may be more practical when used in combination than when interpreted individually, a view which is also shared by the proposed DEA approach. Conceptual mapping has several inherent benefits, including the fact that the involved mapping criteria are made explicit and that a context-specific quality model can be derived for each setting. However, in comparison with DEA, conceptual mapping does not aim directly at benchmarking different systems or projects, since there is no systematic way of combining several factors which might have different measures or scales. A cluster map generated by conceptual mapping requires further analysis by means of visual interpretation that prohibits the automation of the approach. The fact that the approach relies on expert judgment on one hand is an advantage in the sense that prior knowledge and experience is exploited but on the other makes the approach dependent on the availability of sufficiently qualified experts.

In the work by Anda (2007), it is acknowledged that the maintainability of software is affected by a large number of factors, including several code properties but also qualifications of developers, maintenance tasks and tools. Moreover, it is claimed that assessment models focus mainly on individual modules, while the maintainability of complete software systems has received relatively little attention. Therefore, the paper suggests the assessment of maintainability combining structural measures and expert assessments. Two concepts are involved in the interpretation of several metrics (Benestad et al. 2006): Combination, which is the process of providing a combined view of the selected metric values, employs techniques such as the weighted sum or profile comparison. However, combination approaches require the specification of weights, thresholds or coefficients, implying the need for a calibration phase before application. Aggregation refers to the creation of a system-level measure based on class-level measures employing summary statistics such as sums, mean or median values, dispersion or outliers. However, depending on the aggregation approach, different aspects of the underlying information might not be retained. Measures may be combined at the class level and then aggregated to the system level or alternatively, class-level measures may be aggregated before combination.

Significant research work has been devoted to the application of DEA to measure efficiency within an economic or business-related context and even to the software engineering industry (Asmild et al. 2006; Stensrud and Myrtveit 2003). von Mayrhauser et al. (2000) applied DEA to assess the efficiency of 46 software projects from the NASA-SEL database. In this study, outputs also reflect characteristics of the produced software (mainly different types of LOC such as new and modified lines) but inputs are indicators of labor and computer resources (such as technical and management effort, and CPU hours). The model identified which development activities were efficient, defined by the ability of the corresponding organization to produce outputs given input resources. This conforms to the standard definition of efficiency, which refers to the amount of effort that it takes to accomplish a certain task or operation, commonly measured as output/input. In this paper, we illustrated the application of DEA on a software design-related context to investigate whether APIs exhibit improved quality properties compared to applications. The production model in this case identified efficient projects according to a different perspective of efficiency, which refers to the quality of a software system (as captured by software metrics) given the system's functionality and state (as inputs).

DEA can deal with a particularly wide range of problems, some of which have nothing to do with the economic efficiency which has been originally the target of DEA. Some characteristic examples, where inputs and outputs do not have the usual, economic-related meaning, are given next. In a study recommending the relocation of the Japanese capital to a new site, inputs such as "susceptibility to earthquakes" and outputs such as "ability to recover from earthquakes" have been used (Cooper et al. 2005). There are also various innovative engineering applications of DEA, as for instance, the maintenance of highways. One application took into account inputs, such as the climate factor, and outputs, such as the accident prevention factor (Cook et al. 1995). DEA has also been used for the evaluation of a large-scale solar power system compared to fossil, thermal, and nuclear technologies (Criswell and Thompson 1996). A rather unconventional application of DEA was the comparison of baseball players. The only input used was plate appearances, which represent the number of opportunities that the batter had to attempt to produce a walk or a hit. The outputs were the number of walks, singles, doubles, triples, and home runs that the batter produced in those plate appearances (Anderson 2004). A domain which has also attracted the interest of decision makers and researchers is education. In an effort to evaluate the efficiency of school districts, the only input to DEA was the expenditure per pupil, whereas outputs such as pass rates on standardized tests were used (Ruggiero 2004). As it can be observed, there are no limitations in the uses of DEA. What is actually needed is to define efficiency according to the requirements of the specific problem.

9 Conclusions

In this paper, we have attempted to approach the problem of benchmarking object-oriented designs by transferring a tool for performance measurement that is extensively employed in economics. In particular, we have employed Data Envelopment Analysis to obtain relative efficiency scores for a number of open-source libraries and applications. The advantage of DEA is that benchmarking is performed by comparing each software design to its best performing peers rather than a theoretical baseline and that efficiency is estimated by considering all input and output items enabling the comparison of projects with diverse size characteristics.

The vehicle for illustrating the applicability of DEA in the context of software design is the investigation of whether libraries exhibit a superior design quality compared to applications. To this end, a set of widely acknowledged design principles that are expected to underlie the design of libraries has been analyzed. Metrics that reflect the conformance to these principles have been used as outputs in DEA. The results of the application of DEA on twenty-two open-source libraries and twenty-two open-source applications confirm the belief that libraries excel, at least within the context of our study, since their average efficiency score is higher than that of applications. Although the set of inputs-outputs that has been employed refers to particular aspects of design quality, limiting the possibility to generalize these findings, DEA appears to be a promising approach for benchmarking software designs, a task which is not possible when simply examining metric values in isolation.

Further empirical research could focus on investigating the relation of the produced rankings to high-level quality attributes of the examined systems. Moreover, the application of DEA to projects from a single domain in combination with other means of qualitative evaluation could reveal particular projects exhibiting best practices in design.

References

Anda, B. (2007). Assessing software system maintainability using structural measures and expert assessments. In *Proceedings of the 23rd IEEE international conference on software maintenance* (Paris, France, October 2–5, 2007, pp. 204–213). ICSM'07.

Anderson, T. R. (2004). Benchmarking in sports: Bonds or Ruth, determining the most dominant baseball batter using DEA. In W. W. Cooper, L. M. Seiford, & J. Zhu (Eds.), *Handbook on Data Envelopment Analysis* (pp. 443–454). Boston, MA: Kluwer.

Arisholm, E., & Sjøberg, D. I. K. (2004). Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. *IEEE Transactions on Software Engineering, 30*(8), 521–534.

Asmild, M., Paradi, J. C., & Kulkarni, A. (2006). Using Data Envelopment Analysis in software development productivity measurement. *Software Process Improvement and Practice, 11*(6), 561–572.

Banker, R. D., Charnes, A., & Cooper, W. W. (1984). Some models for estimating technical and scale inefficiencies in Data Envelopment Analysis. *Management Science, 30*(9), 1078–1092.

Basili, V. R., Caldiera, G., & Rombach, H. D. (1994). Goal question metrics paradigm. In J. J. Marciniak (Ed.), *Encyclopedia of software engineering* (Vol. I, pp. 528–532). New York: Wiley.

Benestad, H. C., Anda, B., & Arisholm, E. (2006). Assessing software product maintainability based on class-level structural measures. In *Proceedings of the 7th international conference on product-focused software process improvement* (Amsterdam, Netherlands, June 12–14, 2006). PROFES'06.

Bloch, J. (2006). How to design a good API and why it matters. *Companion to the 21st ACM SIGPLAN symposium on object-oriented programming systems, languages, and applications* (Portland, Oregon, USA: ACM Press, October 22–26, 2006). OOPSLA'2006.

Bloch, J. (2008). *Effective java* (2nd ed.). Boston: Addison-Wesley.

Bowman, M., Briand, L. C., & Labiche, Y. (2007). Multi-objective genetic algorithms to support class responsibility assignment. In *Proceedings of the 23rd IEEE international conference on software maintenance* (Paris, France, October 2–5, 2007, pp. 124–133). ICSM'07.

Briand, L. C., Daly, J. W., & Wüst, J. K. (1999). A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering, 25*(1), 91–121.

Charnes, A., Cooper, W. W., Lewin, A. Y., & Seiford, L. M. (Eds.). (1995). *Data Envelopment Analysis: Theory, methodology and applications*. Boston: Kluwer.

Charnes, A., Cooper, W. W., & Rhodes, E. (1978). Measuring the efficiency of decision making units. *European Journal of Operational Research, 2*(6), 429–444.

Coelli, T. J., Prasada Rao, D. S., O'Donnell, C. J., & Battese, G. E. (2005). *An introduction to efficiency and productivity analysis*. Berlin: Springer.

Cook, W. D., Kazakov, A., & Roll, Y. (1995). On the measuring and monitoring of relative efficiency of highway maintenance patrols. In A. Charnes, W. W. Cooper, A. Lewin, & L. M. Seiford (Eds.), *Data Envelopment Analysis: Theory, methodology and applications* (pp. 195–210). Norwell, MA: Kluwer.

Cooper, W. W., Seiford, L. M., & Tone, K. (2005). *Introduction to Data Envelopment Analysis and its uses: With DEA-solver software and references*. New York, USA: Springer.

Cooper, W. W., Seiford, L. M., & Tone, K. (2007). *Data Envelopment Analysis: A comprehensive text with models, applications, references and DEA-solver software*. New York, USA: Springer.

Criswell, D. R., & Thompson, R. G. (1996). Data Envelopment Analysis of space and terrestrially based large commercial power systems for earth: A prototype analysis of their relative economic advantages. *Solar Energy, 56*(1), 119–131.

Dyson, R. G., Allen, R., Camanho, A. S., Podinovski, V. V., Sarrico, C. S., & Shale, E. A. (2001). Pitfalls and protocols in DEA. *European Journal of Operational Research, 132*(2), 245–259.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Boston: Addison-Wesley.

ISO/IEC. (1991). International standard ISO, IEC 9126, International Organization for Standardization, Geneva.

Li, W., & Henry, S. (1993). Object-oriented metrics that predict maintainability. *Journal of Systems and Software, 23*(2), 111–122.

Liskov, B. (1988). Data abstraction and hierarchy. *SIGPLAN Notices, 23*(5), 17–34.

Lorenz, M., & Kidd, J. (1994). *Object-oriented software metrics*. Upper Saddle River, NJ: Prentice Hall.

Martin, R. C. (2003). *Agile software development: Principles, patterns and practices*. Upper Saddle River, NJ: Prentice Hall.

Meyer, B. (2000). *Object-oriented software construction*. Upper Saddle River, NJ: Prentice Hall PTR.

O'Keeffe, M., & O'Cinneide, M. (2006). Search-based software maintenance. In *Proceedings of the 10th European conference on software maintenance and reengineering* (Bari, Italy, March 22–24, 2006). CSMR'06.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM, 15*(12), 1053–1058.

Riel, A. J. (1996). *Object-oriented design heuristics*. Boston: Addison-Wesley Professional.

Ruggiero, J. (2004). Performance evaluation in education: Modeling educational production. In W. W. Cooper, L. M. Seiford, & J. Zhu (Eds.), *Handbook on Data Envelopment Analysis* (pp. 323–348). Boston, MA: Kluwer.

Seng, O., Stammel, J., & Burkhart, D. (2006). Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the 8th annual conference on genetic and evolutionary computation* (Seattle, WA, July 8–12, 2006). GECCO'06.

Stensrud, E., & Myrtveit, I. (2003). Identifying high performance ERP projects. *IEEE Transactions on Software Engineering, 29*(5), 398–416.

Tulach, J. (2008). *Practical API design: Confessions of a java framework architect*. APress.

von Mayrhauser, A., Wohlin, C., & Ohlsson, M. C. (2000). Assessing and understanding efficiency and success of software production. *Empirical Software Engineering, 5*(2), 125–154.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2000). *Experimentation in software engineering: An introduction*. Boston, MA: Kluwer.

Yamashita, A. F., Anda, B., Sjøberg, D. I. K., Benestad, H. C., Arnstad, P. E., & Moonen, L. (2009). Using concept mapping for maintainability assessments. In *Proceedings of the 3rd international symposium on empirical software engineering and measurement* (Florida, USA, October 15–16, 2009). ESEM'09.

Yang, Z. (2009). Assessing the performance of Canadian bank branches using Data Envelopment Analysis. *Journal of the Operational Research Society, 60*(6), 771–780.

Author Biographies

Alexander Chatzigeorgiou is an assistant professor of software engineering in the Department of Applied Informatics at the University of Macedonia, Thessaloniki, Greece. He received the Diploma in electrical engineering and the PhD degree in computer science from the Aristotle University of Thessaloniki, Greece, in 1996 and 2000, respectively. From 1997 to 1999, he was with Intracom, Greece, as a telecommunications software designer. His research interests include object-oriented design, software maintenance, and metrics. He is a member of the IEEE.

Emmanouil Stiakakis is a lecturer in Digital Economics at the Department of Applied Informatics, University of Macedonia—Thessaloniki—Greece. He holds a BSc in Mechanical Engineering from the Aristotle University of Thessaloniki, an MSc in Manufacturing Systems Engineering from Cranfield University—UK, and a PhD in Applied Informatics from the University of Macedonia. His research interests include production and operations management, Total Quality Management, e-business, and digital economy. His research has been published in international journals and conference proceedings.