




Opportunity Costs in Free Open-Source Software

Siim Karus^(✉) 

University of Tartu, 51009 Tartu, Estonia
siim.karus@ut.ee

Abstract. Opportunity cost is a key concept in economics to express the value one misses out on when choosing one alternative over another. This concept is used to explain rational decision making in a scenario where multiple mutually exclusive alternative choices can be made. In this paper, we explore this concept in the realm of open-source software. We look at the different ways for measuring the cost and these can be used to support decisions involving open-source software. We review literature on opportunity cost use in decision support in software development process. We explain how the opportunity cost analysis in the realm of open-source software can be used for supporting architectural decisions within software projects. We demonstrate that different measures of costs can be used to mitigate problems (and maintenance complexity) arising from the use of open source software, allowing for better planning of both closed-source commercial and open-source community projects alike.

Keywords: Code churn · What-if analysis · Scenario analysis · Coding effort · Alternative cost · Opportunity cost · Software cost

1 Introduction

A large subdivision of software engineering research is focused on software estimation. Software estimation is concerned with devising and validating models that can be used to predict or estimate some aspect of software. These estimation models can be used for detecting trends in software development or to support decision making process or planning processes in software development. As such, the aim of many of these models is to provide decision makers with comparable values for different alternatives. Despite much effort put into evaluating and improving the accuracy of different models [1], the applicability of these models for actual comparison of alternatives is often overlooked. Hereby, we aim to collect the different approaches to measure cost in software development and review the approaches' applicability for comparing alternative actions.

In order to accomplish the set task, we perform an analytical review of objective comparison options. That is, we give emphasis on data-driven options instead of subjective expert (or community) opinion or evaluation. As a limitation, the options are only readily applicable on open source software, which makes relevant data available. We start by reviewing options for estimating the cost of software development. This is then complemented with an overview of literature reporting opportunity cost or alternative cost use in software engineering. In particular, we look for the following aspects:

1. What is the measure of the cost used?
2. Is the estimation model objective? That is: can the estimation be made solely based on the data?
3. Does the literature confirm the generalisability of the estimation model on different projects?
4. Does the literature report the confidence and accuracy of the model?

The first aspect is for classification of the approaches and identification of different measures of cost used in software engineering. The second and the third aspect describe the models' independence from specialized knowledge of experts that might not be available or have comparable skill level to the experts used in the original studies. As such, objective data-based models that have been evaluated on multiple projects are preferred due to their wider applicability and higher theoretical reproducibility. The last aspect evaluates whether the model can be used for comparison of alternatives in general. It is important to know the accuracy of the estimations in order to assess, whether two estimations are significantly different from one another. If two estimations differ less than the estimation error at given confidence level (or, to a limit, to given p-value), then the estimations can not be considered significantly different from each other.

The literature overview is followed by examples of how decision making in or involving free open-source software setting can benefit from the comparison of alternatives based on the estimation models. We highlight the challenges of extending the process of comparing alternatives to closed-source commercial software development (as opposed to FOSS) and list the main limitations of opportunity cost analysis. Finally, we conclude the study by listing opportunities for the future.

2 Measures of Cost

Cost is an amount that has to be paid or given up in order to get something [2]. In business, cost is usually a monetary valuation of (1) effort, (2) material, (3) resources, (4) time and utilities consumed, (5) risks incurred, and (6) opportunity forgone in production and delivery of a good or service [2]. In software projects, cost can be measured using several units. In commercial environments, the value of different goods is made comparable by introducing a conversion rate for all goods into a common good - currency. Consequently, currency is commonly used to evaluate costs.

2.1 Monetary Value

Even though the monetary value is most commonly used for cost measurement, it has many variations. A contemporary good cost estimation model is supposed to take into account any development and delivery effort until a release of the software [3]. Sometimes the monetary cost is also considered to include sales effort and expenses, efforts to adjust to changes, maintenance efforts, operational efforts and expenses, brand-related turnover, or other expenses or revenue opportunities. As such, the different monetary cost estimation methods can lead to very different estimations and need to be evaluated separately.

In free open-source software development, the monetary cost is usually of negligible interest, as the product is not sold for profit. In some extreme cases the projects may be sponsored, in which case the monetary dimension comes to play as the product needs to keep the sponsors interested and the sponsorship is an opportunity to allow higher expenses in the project.

2.2 Effort (Person-Hours)

As noted before, cost is usually measured in monetary evaluation. However, cost can be measured using other metrics as well. As the cost of workforce is highly volatile, a proxy of monetary cost, effort in worktime (usually person-hours or person-months) is used instead [1]. This measure is much more stable as the cost is measured directly and therefore does not differ as much in time and space as currency [3]. Nevertheless, such cost is highly dependent on the individual assigned to the task, as different developers can need different amount of time to complete the same task. To make the situation even more complicated, reliable measuring of time spent on task is very difficult. As software development requires planning and substantial mental work, it is not limited to a location or registered by automatic means. Consequently, time measurements are often based on self-reporting of the developers, which is not very accurate.

2.3 Time (Calendar)

The release cycle and release timings have become an important aspect of software development. It is the need to publish early that competes with the time available for development and testing [4, 5]. Consequently, the value measured by the cost can be the earliness of the next release. The cost does not have to be linear or even monotonic in relation to calendar time as certain dates or weekdays can be considered more valuable or less valuable than others.

2.4 Size

Sometimes the amount of code is used as a measure of cost. The size reflects the amount of effort put into developing the software – at least in the actual writing effort. Nevertheless, sometimes smaller size can be more valuable as it means less code to go through and to understand. As such, the size of the software can have a positive or negative value depending on the evaluator's objective.

2.5 Code Churn

Code maintenance effort is often evaluated based on code churn. Whilst code churn can be measured in several different units from modules and files to tokens and symbols, it is most commonly measured in lines of code (LOC). Code churn in lines of code is calculated as the sum of the number of lines of code added, modified and deleted between two revisions of a software module [6]. Some studies only count lines that are written in certain programming language and affect program logic. However, all source code needs to be maintained.

In addition to providing an indicator of code maintenance effort, code churn has also been shown to be correlated with software defects [7, 8]. In those cases, code churn is usually aggregated over a certain period of time (commonly a year [9–11]).

Naturally, code churn estimates over a certain period of time show how stable is the code-base. Projects with stable code-base can be considered more mature as there is less on-going development and fewer bug fixing going on. Thus, in situations where high reliability and low maintenance costs are important, projects with low code churn estimates would be generally preferable. Of course, this is on the premise that the stability of the project is not confused with the abandonment of the project.

Code churn is often estimated based on object-oriented metrics of the project [9, 12, 13]. Nevertheless, process-based metrics and organisational metrics have been shown to provide far better results [14–16].

2.6 Other Cost Measures

In addition to these more common cost measures, sometimes the relevant and substantial costs can be of very specific or difficult to measure kinds. For example, costs of brand, morale or public opinion can be detrimental. When discussing technical debt, effects towards monetary cost are always to be considered together with potentially longer term effects on morale, productivity, risk, and quality [17]. The practical limitation for uniform opportunity cost assessment is that some of these costs (e.g. morale) are very difficult to quantify. In those cases, the opportunity cost evaluations will need to be multi-dimensional in order to take account of all the different costs and scales they can be measured on.

3 Opportunity Cost in FOSS

The comparison of alternative actions requires the valuation of the outcomes from making a decision. The basis for such rational value-based decision-making in economics is opportunity cost. **An opportunity cost** (also known as **alternative cost**) is a benefit, profit, or value of something that must be given up to acquire or achieve something else [2]. That is, an opportunity cost can be expressed as:

$$\text{Opportunity cost} = \text{value of an option not chosen} - \text{value of chosen option.}$$

Table 1. Search results and topics. Search results shows ‘number of results from keywords “opportunity cost” “software”’ + ‘number of results from keywords “alternative cost” “software”’.

Database	IEEE Xplore	ACM Digital Library	ScienceDirect	Total
Search results	13 + 4	8 + 3	51 + 11	90
Software engineering articles	10	6	19	35
Relevant articles	4	2	2	8
Topics	Maintenance, testing, planning	Testing, planning	Testing, planning	
Measures	Monetary, effort	Monetary	Monetary	
Data-based articles	1	0	0	1

In rational value-based decision-making, the chosen option should always be the highest-valued option. As such, one can look at decision-making as an optimisation process.

Even though the formula is simple, the evaluation of alternatives rarely is. The main complexity stems from the fact that the “value” of an alternative is often difficult to define precisely. The cost estimation models are generally focusing just a few of the measures of the true value of an option (e.g. some models try to estimate monetary gain or loss, others focus on delivery times), which might not even be directly relatable to one-another. Consequently, if multiple factors apply the optimal decision can be found using Pareto optimisation [18]. This also means that the best option may have sub-optimal value in some respect. For example, faster delivery date may be more important than lower monetary expense for some projects.

We searched three major repositories of research relating to software engineering: IEEE Xplore¹, ACM Digital Library² and ScienceDirect³. As keywords, we used “opportunity cost” and “alternative cost”. The search was refined by adding a keyword “software” to both searches to eliminate the bulk of studies not relating to software. Nevertheless, the results still contained largely studies that did not discuss topics relating to software development as studies discussing opportunity cost in other fields commonly listed the software packages used for opportunity cost estimation and comparison. In summary, the search from the databases yielded 90 results with only 35 relating to software development processes⁴. The 55 results were rejected based on the review of titles and abstracts, which revealed these to be discussing opportunity costs in agriculture, medicine, (non-software) economics, or other fields. The remaining 35 papers were manually reviewed to filter out papers where opportunity costs were discussed only in reference (e.g. as future work, introduction or only in abstract) or did not relate to decision-making in any way (e.g. algorithm optimisation). This left us with eight papers (four from journals, four from proceedings with the earliest from 1990 and latest from 2016) relevant to our current focus. The review also identified additional related keyword “options analysis”, but searching using this keyword did not reveal additional articles of relevance to our study. The statistics of the search results can be found in Table 1.

We found three studies discussing opportunity costs in the planning of software development activities: [19, 20], and [21]. All of these studies were conducted in commercial environment and had monetary cost included in the analysis. Spinola et al. defined indexes that can be used for balancing effort with monetary cost when assessing resource (developers) reallocation options [19]. Whilst the paper shows a nice

¹ <https://ieeexplore.ieee.org/>.

² <https://dl.acm.org/>.

³ <https://www.sciencedirect.com/>.

⁴ We also looked into Google Scholar search, but found its tools for filtering the results too limiting for practical use. The original search keywords would get about 40,000 hits due to the popularity of “opportunity cost” in many fields. Unfortunately, Google Scholar does not provide means to limit search to specific fields and by adding exclusion keywords for more frequent non-related fields, we could limit the search results down to about 4000 before hitting the limitations of Google Scholar query complexity.

example of how to derive the indexes needed for cost computation and comparison, the practical application of the analysis process would require expert input in order to identify appropriate index values. Similarly, Cai et al. proposes a framework that can be used to justify architectural decisions by minimising the risks and monetary costs of potential changes [21]. Özogul et al., on the other hand, presents a case study of a formula used to evaluate alternative investments in a hospital information system development and operation [20]. The investment value estimation is based on sub-estimates from experts or past experience.

Papers relating to software testing were the most common among the software engineering papers on opportunity cost: 4 out of 8 studies looked at aspects of testing. Two of these were trying to answer the question of when should one stop testing and release the software: [4] and [5]. Both of these looked at monetary cost balance of the cost of testing (and fixing bugs found during testing) and the cost of resolving bugs in released software. In [4], the authors try to improve a previously developed model for assessing the optimal time to stop testing by adding measures to handle uncertainty of the values of some components of the model. In overall, their model is robust but rather simplified with only the efficiency of the testing team (that is, how fast do they detect bugs) left to the experts to evaluate. [5] offers a more detailed model for calculating the costs and introduces the concept of patching to the model. The other two articles look into prioritization of tests in order to balance testing expenses with the risks from missed bugs [22], and finding the balance between the number of automated test cases and the number of human test executions [23].

Finally, one paper discussed opportunity cost in software maintenance: [24]. In there, the authors were looking at the opportunity costs from the inner-company client's perspective. They concluded that in their case study the costs for the IT department were not affected by the time bugs remained open (the "lead time"). However, they did find that it does influence the opportunity costs of the users of the information systems and could lead to sub-optimal distribution of costs within a company. As the analysis was performed on actual historic data, it does not rely on the availability and skill of experts, which makes it stand out of all the papers on opportunity cost reviewed here.

Even though "opportunity cost" is a term rarely used in current software engineering research, a properly reported study of "cost estimation" can be used for assessing opportunity costs as well. This is an advantage as "cost estimation" is a popular topic with a combined total of over more than 2000 search hits in the three databases used in this study.

4 Opportunity Cost in Decision Making

Opportunity cost can be used in several different decision-making scenarios. For example, it is useful for picking the platforms or libraries for use in a project potentially avoiding the need to reverse such decisions (e.g. reverting from Angular JS due to maintenance costs incurred by its fast pace of changes); or evaluation of options for refactoring a project's component and/or dependency structure. Hereby we give examples on how opportunity cost analysis can be used in these scenarios depending on the available cost measure.

4.1 Choosing a Product

Let us assume a scenario where a project manager needs to choose, which software library or framework to use in the project. There are several approaches to solving this task depending on the aim of the project. Given two or more alternative options, the project manager can make a preliminary choice of one library that seems most suitable (alternatively, he could choose to not use a third party library). Based on the choice, the project manager can calculate the costs of adopting alternative libraries. If the opportunity cost of adopting an alternative library is favourable, the project manager can change the choice to become the better alternative.

The monetary opportunity cost should be easy to understand – positive opportunity cost means that the monetary gains from adopting the alternative library outweigh the gains from the current choice. Accordingly, negative monetary opportunity cost favours current choice.

The opportunity cost in effort (person-hours) is a bit more complex to act upon. As effort is by its nature a loss (the team spends effort) and the opportunity cost is defined as difference of gains, the opportunity cost in effort is the negative of the difference of the team's efforts. Thus, a positive opportunity cost in effort means that the alternative requires less work than the current choice. This means that if the project is mostly voluntary, the minimisation of effort might be preferred (thus, the choice should be changed if opportunity cost is positive). In some cases, both the effort and monetary costs need to be considered together as separate dimensions. This can be a case if there are license or infrastructure costs linked to the adoption of libraries. We highlight that the comparison is not linear as the difference in monetary cost can be insignificant if the budget is high or the monetary cost difference can be limiting due to budget limitations. Thus, the monetary cost component tends to behave in a stair-like fashion rather than linear fashion.

The opportunity cost in time is even more complicated to calculate, as the value of the timing of actions is rarely monotonic. For example, optimal release opportunities tend to depend on the weekly and monthly cycles. In addition, special external events or deadlines from clients may make certain periods (e.g. releasing consumer-oriented services ahead of major holidays) more favourable than others. The same applies to the team members' ability to react to other team members' changes in the source code.

As mentioned in the description of the measure of size, size can be either a positive or a negative attribute of the software. Consequently, it needs to be handled accordingly when calculating the opportunity cost.

As code churn is an indicator of maintenance effort, it can provide interesting insights into the quality and development stage of the library. A high code churn estimation will indicate that the library will be actively developed or maintained. A low estimation means that the module is either stable or abandoned. Consequently, when looking to adopt a more stable library, one would prefer libraries with lower code churn estimation (code churn is a negative feature). If one would like to use more changing and agile libraries, high code churn would be preferred (code churn is a positive feature). In general, commercial companies focused on reliability are more interested in using stable libraries and frameworks in their projects [25].

This example of choosing a library or a framework can be extended to a scenario of choosing a software application from multiple alternatives. For example, software with lower estimated code churn can be expected to be more stable than other alternatives.

4.2 Refactoring Options

Software component maintainability can be an important factor when deciding how to refactor the software. The refactoring options can include replacement of a component, integration of a component into other source code, or preservation and continued maintenance of the component. Using a framework for assessing a module's or component's quality (e.g. via a method proposed by Upadhyay et al. [26]), will give general quality assessment on the module, but does not give an accurate assessment of the cost of replacing or integrating the component as the maintenance of software components is highly dependent on one-another (as shown by Mari et al. [27]). Understanding this, we can devise a following method for assessing the alternative cost of a software module.

Let us assume that a project manager or architect is wondering whether certain sections of the software should be replaced or removed from the project. We can find out, whether removing (or replacing) the modules has positive effect on the maintenance of the software by simulating the removal (or replacing) of the module and estimating the opportunity cost in maintenance effort.

From the general definition of opportunity cost, we construct the definition of opportunity cost of maintaining a software module. Opportunity cost or alternative cost of not having a module m in project p (A_m^p) is the difference between the cost of maintaining or developing the project without the module (E_m^p) and the cost of maintaining or developing the project with the module (C^p).

$$A_m^p = E_m^p - C^p$$

It is clear from the definition, that in order to calculate the opportunity cost of maintenance of the project without a module, we need to know both the cost of developing the project with and without a module. These two costs never occur in the same project at the same time, which rules out using actual data for calculating the opportunity cost. Therefore, we need to use an oracle that would tell how much the development in one or another scenario would cost. The part of the oracle can be performed by a cost estimation model, which allows reproducible objective analysis of options, or by experts (or a combination of experts and cost models), which is less generalizable.

Having found out that the opportunity cost of removing the module is significantly⁵ negative, we know that we should keep the module. If the opportunity cost would have turned up significantly positive, you would know that the module is safe to remove. If the opportunity cost is near zero, the removal of the module is likely to have little to no effect on the maintainability of the project. Similarly, one can simulate any refactoring

⁵ Significantly means that the probability of being mistaken is less than acceptable by your confidence threshold. Commonly confidence of 0.95 or higher is used meaning the probability of being mistaken is 5% or less.

option and calculate corresponding opportunity cost. Therefore, it would be possible to evaluate all refactoring options before taking action.

As the possible simulations are not limited by anything but the oracle's ability to sense the changes, the same process can be used for more accurate evaluation of implementation alternatives. The oracle has another limitation – one needs to know whether the oracle's accuracy is below or above the significance threshold (confidence) of the developer. Thus, only estimation models with published error and confidence (or p-value) can be used as an oracle.

4.3 Commercial Applications

In commercial applications, the monetary cost of the project becomes more relevant than in free software. The opposite is often true for the effort as the availability of workforce is less volatile. Simplistically, the monetary value can be assumed to stem from effort alone [1]. In practice, this can be an oversight as depending on the project, the infrastructure costs or even public image can have higher impact on the monetary costs than developer salary. Commercial evaluation also needs to take into account inflation, which affects monetary costs [20].

On the other hand, a more rigid structure of commercial projects will allow more precise cost estimation methods to be used for opportunity cost estimation. This can lead to much more accurate opportunity cost analysis and better options for finding the optimal behaviour. In this study, we focus on free software as the data on free software has better availability and lends itself to easier replication of the study.

5 Limitations of Cost Estimation

An important factor that was highlighted by Özogul et al. is that the value of the measurement unit changes in time [20]. As they were studying the monetary costs, they resolved that limitation by calculating the present value of any future values. They could do that on the assumption of knowing the approximate average inflation rate for the prices in the studied currency. In general, it is not that simple to compare values or costs from different time periods. This also applies to code churn as the efficiency of software developers is increasing due to the introduction of frameworks, higher level languages and design-time code generation. However, we are currently not aware of any standard methods or discounting the effects of such advancements in code churn analysis.

As all estimations, cost estimation are not ideal and exact. Thus, one needs to consider the limitations of the accuracy of cost estimation models used. We highlighted that as a requirement for any cost estimation model to be generally useable as it is possible to identify significant cost differences only with models that have known error ranges and confidence levels. Not understanding and controlling for the significance of difference can easily lead to invalid conclusions. As in our experience similar software modules often have similar cost estimations, this can lead to project managers switching software modules and components too often (even when there is no significant benefit of cost) causing more cost to incur due to instability of the project architecture.

6 Conclusions and Future Perspectives

Opportunity cost is a key concept in economics when describing rational decision-making. A literature search of opportunity cost in software in three larger software-related research databases found only a handful of relevant results. Therefore, it is clear that the value-based decision making rationales in software engineering are not well established. Even more, we found that only 1 of 8 papers demonstrated opportunity cost calculation without relying on expert opinion. Only 1 out of 8 papers claimed that the approach used was tested on more than 2 projects and none of the studies claimed verification from more than 2 experts when expert opinion was needed. As such, the generalisability of the current studies is weak. None of the studies reported the accuracy and confidence of the proposed methods.

On positive note, we found that there is at least one cost estimation method that can be used for opportunity cost analysis. Furthermore, the identified method uses a model that does not require any expert input – the estimations were based solely on the data available in a source code repository history. This allows the approach to be used even in situations where experts with sufficient competence are not available without needlessly exposing the estimation process to a potential human error. The technique presented in this paper shows how the effects of decisions made during software development process can be simulated. More specifically, we offer an approach for better identification of the drivers of software coding effort in open source software projects. As such, we cater for the future-oriented needs of software development analysts as defined by Buse et al. [28]. As an added benefit of using easy-to-use end-user-oriented analysis platform, our analysis technique can be easily deployed without deep understanding of statistics, databases or software source code management systems.

The obvious future perspective is the need for more studies of opportunity cost modelling in software engineering. Some of the scarcity of opportunity cost modelling can be due to the performance of the estimation models, which we have seen to be sufficient for limited success. Improving the estimation abilities of the models can lead to significantly better and more detailed what-if analysis.

Acknowledgement. This work was supported by the Estonian Research Council (grant IUT20-55).

References

1. Jørgensen, M., Shepperd, M.J.: A systematic review of software development cost estimation. *IEEE Trans. Softw. Eng.* **33**(1), 33–53 (2007)
2. BusinessDictionary. <http://www.businessdictionary.com/>
3. Pandey, P.: Analysis of the techniques for software cost estimation. In: *Proceedings of 2013 Third International Conference on Advanced Computing and Communication Technologies*, pp. 16–19. IEEE, Rohtak (2013)
4. Dalal, S.R., Mallows, C.L.: Some graphical aids for deciding when to stop testing software. *IEEE J. Sel. Areas Commun.* **8**(2), 169–175 (1990)

5. Kapur, P., Shrivastava, A.: Release and testing stop time of a software: a new insight. In: Proceedings of 4th International Conference on Reliability, Infocom Technologies and Optimization. IEEE, Noida (2015)
6. Hall, G.A., Munson, J.C.: Software evolution: code delta and code churn. *J. Syst. Softw.* **54**(2), 111–118 (2000)
7. Munson, J.C., Elbaum, S.G.: Code churn: a measure for estimating the impact of code change. In: Proceedings of International Conference on Software Maintenance (ICSM), pp. 24–31 (1998)
8. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Proceedings of the International Conference on Software Engineering, pp. 284–329 (2005)
9. Thwin, M.M., Quah, T.-S.: Application of neural networks for software quality prediction using object-oriented metrics. *J. Syst. Softw.* **76**(2), 147–156 (2005)
10. Koten, C.V., Gray, A.R.: An application of Bayesian network for predicting object-oriented software maintainability. *Inf. Softw. Technol.* **48**(1), 59–67 (2006)
11. Karus, S., Dumas, M.: Predicting coding effort in projects containing XML. In: Proceedings of 16th European Conference on Software Maintenance and Reengineering (CSMR), pp. 203–212 (2012)
12. Li, W., Henry, S.: Object-oriented metrics which predict maintainability. *J. Syst. Softw.* **23**(2), 111–122 (1993)
13. Zhou, Y., Xu, B.: Predicting the maintainability of open source software using design metrics. *Wuhan Univ. J. Nat. Sci.* **13**(1), 14–20 (2008)
14. Rahman, F., Devanbu, P.: How, and why, process metrics are better. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 432–441. IEEE Press, San Francisco (2013)
15. Karus, S., Dumas, M.: Code churn estimation using organisational and code metrics: an experimental comparison. *Inf. Softw. Technol.* **54**(2), 203–211 (2012)
16. Nagappan, N., Murphy, B., Basili, V.R.: The influence of organizational structure on software quality: an empirical case study. In: Proceedings of 30th International Conference on Software Engineering (ICSE), pp. 521–530. ACM, Leipzig (2008)
17. Tom, E., Aurum, A., Vidgen, R.: An exploration of technical debt. *J. Syst. Softw.* **86**(6), 1498–1516 (2013)
18. Wierzbicki, A.P.: A mathematical basis for satisficing decision making. *Math. Model.* **3**(5), 391–405 (1982)
19. de Mesquita Spinola, M., de Paula Pessoa, M.S., Tonini, A.C.: The Cp and Cpk indexes in software development resource relocation. In: Portland International Center for Management of Engineering and Technology, pp. 2431–2439. IEEE, Portland (2007)
20. Özogul, C.O., Karsak, E.E., Tolga, E.: A real options approach for evaluation and justification of a hospital information system. *J. Syst. Softw.* **82**(12), 2091–2102 (2009)
21. Cai, Y., Sullivan, K.J.: A value-oriented theory of modularity in design. In: Proceedings of the Seventh International Workshop on Economics-Driven Software Engineering Research, pp. 1–4. ACM, St. Louis (2005)
22. Schwartz, A., Do, H.: Cost-effective regression testing through adaptive test prioritization strategies. *J. Syst. Softw.* **115**, 61–81 (2016)
23. Ramler, R., Wolfmaier, K.: Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In: Proceedings of the 2006 International Workshop on Automation of Software Test, pp. 85–91. ACM, Shanghai (2006)
24. Chan, T.: Beyond productivity in software maintenance: factors affecting lead time in servicing users' requests. In: Proceedings of International Conference on Software Maintenance, pp. 228–235. IEEE (2000)

25. Raemaekers, S., Deursen, A.V., Visser, J.: Measuring software library stability through historical version analysis. In: Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM), pp. 378–387. IEEE, Trento (2012)
26. Upadhyay, N., Despande, B.M., Agrawal, V.P.: Towards a software component quality model. In: Meghanathan, N., Kaushik, B.K., Nagamalai, D. (eds.) CCSIT 2011. CCIS, vol. 131, pp. 398–412. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-17857-3_40
27. Mari, M., Eila, N.: The impact of maintainability on component-based software systems. In: Proceedings of 29th Euromicro Conference, pp. 25–32 (2003)
28. Buse, R.P., Zimmermann, T.: Information needs for software development analytics. In: Proceedings of the 34th International Conference on Software Engineering, pp. 987–996. IEEE Press, Zurich (2012)