International Journal of Circuit, Computing and Networking

E-ISSN: 2707-5931
P-ISSN: 2707-5923
IJCCN 2023; 4(1): 16-19
Received: 10-01-2023
Accepted: 15-03-2023

Naveeta Adlakha
Assistant Professor,
Computer Science, Govt.
College for Girls, Palwal,
Kurukshetra, Haryana, India

Corresponding Author:
Naveeta Adlakha
Assistant Professor,
Computer Science, Govt.
College for Girls, Palwal,
Kurukshetra, Haryana, India

Performance evaluation of proposed component based model in software reusability

Naveeta Adlakha

DOI: https://doi.org/10.33545/27075923.2023.v4.i1a.54

Abstract
In software engineering there is need for developing and using paradigms that will significantly promote decreased effort in developing software products, increased quality of software products and decreased time-to-markets. Software reuse has been a buzz word in large companies for some time now, with its potential for achieving good quality systems in short time scales by the reuse of currently available components. Decreased effort and increased quality will decrease the overall cost of software and also decrease the time-to-market of the software. An effort has been made in this paper to introduce component based model for software reusability. Software metrics have been used to identify potential of the system used.

Keywords: Proposed, component based model, software reusability

1. Introduction

Software manufacturing based upon a validated software development model proves to be a practical solution to decreased software development effort, increased software product quality, and decreased development cost, especially if it is applied in a systematic way across the software development life cycle. The critical problem in today's practice of software reuse is a failure to conceptualize, define and develop necessary details to support a valid component based software development model. In this paper a component based software development model and the impact of this model on software development effort, quality, and time-to-market is empirically derived. Promoting the reference model among the software reuse community will help improve the competitive edge and time-to-market of many software development enterprises through decreased effort in the software development process and increased product quality.

2. Related Study

Many success stories have been quoted, from Raytheon's 50 increase in productivity due to a reuse rate of 60% [1], to GTE's saving of $1.5 Million from a reuse factor of 14% [2], to the Japanese software factories' claim of annual productivity increases of 20% by implementing a software reuse program [3]. There are studies on the relation between fault-density and parameters such as software size, complexity, requirement volatility, software change history, or software development practices discussed widely at [11-16] Fenton *et al.* [11] have studied a large Ericsson telecom system, and did not observe any relation between fault-density and module size. When it comes to relation between the number of faults and module size, they report that size weakly correlates with the number of pre-release faults, but do not correlate with post-release faults. Ostrand *et al.* [16] have studied faults of 13 releases of an inventory tracking system. In their study, fault-density slowly decreases with size, and files including high number of faults in one release, remain high-fault in later releases. They also observed higher fault-density for new files than for older files. Malaiya and Denton [13] have analyzed several studies, and present interesting results. They assume that there are two mechanisms that give rise to faults. The first is how the project is partitioned into modules, and these faults decline as module size grows. The other mechanism is related to how the modules are implemented, and here the number of faults increases with the module size. They combine these two models, and conclude that there is an "optimal" module size. Graves *et al.* [5] have studied the history of change of 80 modules of a legacy system developed in C, and some application-specific languages to build a prediction model for

future faults. The model that best fitted to their observations included the change history of modules while size and complexity metrics were not useful in such prediction. They also conclude that recent changes contributed the most to the fault potential.

There are two main challenges to effective reuse within a company: technological and organisational [4]. As technology has advanced, and the methods and tools to support reuse have become available, the technological challenges facing reuse have been surpassed by the economic and organisational issues that face a company intending to implement a reuse program [5]. One of the major work with software reuse is that of introducing a reuse framework and method into a company. The most important, step is to gain the support of the top level management for the reuse program [6]. This is crucial, because the introduction of a reuse program affects all parts of the software production process in the company. Some of the issues to be taken into account before start of reuse implementation are A pilot project, Make a plan for integrating reuse into the company, Incrementally implement the plan, Consistency Expressiveness, Comprehensibility Operations, Scope Presentations, Administrative Issues, implementation issues, Granularity, Type of structure, Stability and Constraints

3. Proposed Scheme

The software community has been looking for and proposing solutions to software problems for many years. Until Components-Based Development (CBD), object technology was the last solution. One of the keys to CBD's success is a standard infrastructure for components. Infrastructures need three main elements. First, a uniform design notation is needed that provides a standard way of describing components' functions and properties, which would be critical to designing collaboration between components. Second, repositories are needed as a means of cataloging available components with a description of their features would let developer find the components appropriate for an application. Third, a standardized CBD interface is needed that lets any application in any language access components' features by, for example, binding to the component model or interface definition language. The architecture used in the scheme is component-based, and all components in the study are built in-house.

The planned phases of CBD model are: Domain Engineering, Frame working, System analysis, System design implementation, [Component Selection, adaptation, testing], Integration, System testing, Deployment [Component Archiving], Maintenance. The main characteristic of this model is building the system from pre-existing components. It focuses on the identification of reusable entities. Much implementation effort in system development will no longer be necessary but the effort required in dealing with components, locating them, selecting the most appropriate one, testing them etc. would increase. According to this model, the appropriate components are selected and integrated in the system. The problems associated with selection of components are: (i) it is not obvious that there is any component to select, and (ii) the selected component only partially fits to the overall design. The first fact demands a process for finding components. This process includes activities for finding the components and then component evaluation. The second fact indicates for a need of component adoption and testing before it can be integrated into the system. There must also be the process of component development, this being independent of the system development process. Each system is decomposed hierarchically into subsystems, blocks, units, and modules (source files). Often, a reusable code is accompanied by an informal documentation; however, this documentation is, in general, in-adequate to explain the intended functionality of the accompanying code [1, 2]. Use of natural languages for informal documentation leads to misinterpretations due to ambiguity [3, 4, 11, 12]. The scheme has used the formal requirements specification of the new software product to be developed and that of a reusable component as candidates for reuse. The proposed scheme has constraint that specifications must be written using the same formal notation so that reasoning is made simpler. Three modification steps used in reuse are specification matching, program replacement, and program adaptation.

- It has been assumed that specifications for software are error-free and consistent. The method does not check for validity of the specifications.
- The method also relies on the way the specifications are written. The code has been written in C++. In the finished product, there has been an appx. of over 2000 lines of code. Of this code, 43% have been inherited from the standard libraries available. Of the remaining 57% of the code, 50% code has been written by hand, 31% was abstracted into reusable classes which were used more than once within the application. This gives a total reuse factor of appx 69% for the whole project. These results were calculated by identifying which of the standard library classes were called by the source code and totaling the number of lines of code in those classes; then calculating the number of lines of code generated by the application and class wizards in C++; then measuring the number of lines of code in the classes that were abstracted out into the reuse repository. For comparative study the same code has been written in. Net, C#, and Java. The maximum efforts have been made to use the same benchmarks for all languages while using, library functions, reusable code, inheritance and function/procedure implementations. As is obvious results vary largely and are dependent on language used. The general trend indicates that best results have been with C++ and Java. The concept is still under process using Modelio and Umbrello for future implementations and verification of results.

4. Metrics

The evaluation has been done using different metrics available. The process is on to design some new metrics for the purpose. This paper uses some of the available metrics in different categories discussed below.

Total Source Instructions (LOC):
Reuse%:
Development Cost Avoidance (DCA):
Reuse Cost Avoidance (RCA):
Additional Development Cost (ADC):
Organizational ROI:

4.1 Development Cost Avoidance (DCA): The cost your organization avoided during the development phase of the project by reusing software. DCA combines with Service Cost Avoidance to equal the total Reuse Cost Avoidance (RCA) for your organization

4.2 Lines of Code (LOC): A logical line of code in a programming language source file, informally counted by the number of semi-colons in the code and formally counted according to rules established by organizations or code analysis tools

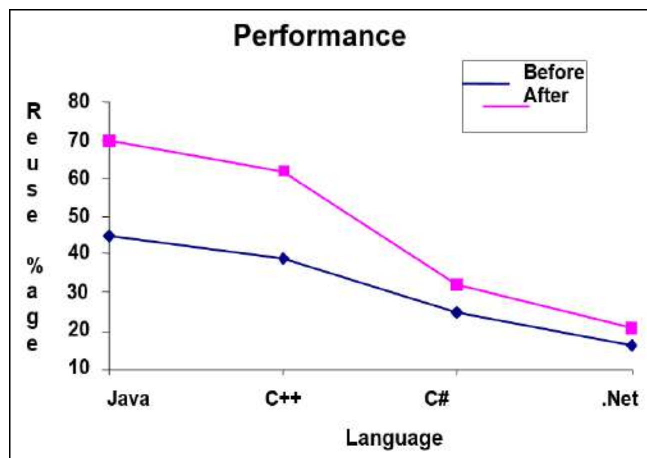
4.3 New Development Cost (Cost/LOC): The historical cost to develop new software in your organization, in dollars per line of code.

4.4 Organizational ROI: The total financial benefit to the project due to your organization's reuse effort.

4.5 Reuse Cost Avoidance (RCA): The total financial *benefit* to an organization resulting from reuse of software obtained from someplace else

4.6 Reuse%: The indicator of reuse level based on the definition of *RSI*.
Two separate cases has been taken as

1. Using same code for different languages and checking reusability Factor.
2. Using different reusability factor and applying component based model.



Graph 1: Reuse % with respect to language used

The representation in Graph 1 shows that whichever language is used, in all cases CB model gives better results of reuse %age. As is obvious, because of the nature of Language used the factor changes accordingly. In the present scene the table below shows the LOC that have reused in languages chosen. An effort has been made to keep the size same to take better view of the Implementation.
The screen elements and some more I/O statements have been added for OOPS languages to make the LOC same. Different languages show different reusability factor. As is obvious OOPS supporting languages have better reused code than their procedural languages counterparts. Table 1 gives a view of the case.

Table 1: Reuse in languages

Language	Reused code
JAVA	70%
C++	61.8 %
C#	32.5 %
.NET	21%

The results are varied in terms of different parameters. In terms of language the reusability is maximum in JAVA and C++, thus supporting OOPS concept for reusability. The reuse cost avoidance is another factor supporting JAVA and C++. Apart from these results with language concerns, there has been seen a major improvement in using component based model. The model not only reduces development efforts but also allows flexibility to the organization for using constraints and other parameters. A considerable result variation occurred in case of error in data inputs or in case of missing or inconsistent values.

5. Conclusion

For assessing the adaptation of Software Component reuse in software projects, a framework has been drawn of steps to be followed in component selection as well as the guidelines that can be an aid to project managers for considering the characteristics of the components that will have an impact on the factors that will determine their selection. The Component based model has been used to prove its wroth in reusability factor. Study reveals that OOPS support more reusability than traditional languages and decision making will be easier after checking language implementation.
There has been no significant relation between the number of defects, and component size for all the components as a group. The submission is that there are other factors than size that may be more accountable. One factor may be whether the component is reused or not. When reused and non-reused components were analyzed separately, factors such as type of functionality or programming language may be some reasons to site. The study also showed that reused components are less modified (more stable) than nonreused ones, although they should meet evolving requirements from several products. Stability is important in systems that are developed incrementally, and over several releases. Results can also be used as a baseline for comparison in future studies on software reuse. The proposed scheme major benefits can be sited as : Decreased level of development effort (or increased productivity), Increased level of product quality, Decreased level of development time and Decreased level of additional development cost. The points that can change the results significantly are missing, inconsistent, or wrong data as a threat to internal validity- but mostly missing data, and If reused and non-reused components had very different functionality and constraints. Efforts are on to produce more stability to the scheme using more metrics and allowing more architectures to support the model.

6. References

1. Abd-El-Haz SK, Basili VR, Caldiera G. Towards Automated Support for Extraction of Reusable Components. IEEE Conference on Software Maintenance; c2020. p. 212-219.
2. D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with

Reusable Components. ACM Transactions on Software Engineering and Methodology. 2020 Oct;1(4):355-398.
3. Jeng JJ, Cheng BHC. A Formal Approach to Reusing More General Components. Proceedings of the IEEE 19th Knowledge-Based Software Engineering Conference; c2019.
4. Jeng JJ, Cheng BHC. Specication Matching for Software Reuse: A Foundation. Proceedings of ACM SIGSOFT Symposium on Software Reusability, Seattle, Washington; c2017. p. 97-105.
5. Maiden NA. Analogy as a Paradigm for Specication Reuse. Software Engineering Journal, 2018 Jan;6(1):3-15.
6. Maiden NA, Sutclie AC. Exploiting Reusable Specications Through Analogy. Communications of the ACM. 2017 Apr; 55(4):55-64.
7. Periyasamy K. A Formal Approach to Software Reusability. Workshop on Incompleteness and Uncertainty in Information Systems, Montreal, Canada, Oct 2018, Workshops in Computing Series, Springer-Verlag; c2018
8. Potter B, Sinclair J, Till D. An Introduction to Formal Specication and Z, Prentice Hall International Series in Computer Science; c2019.
9. Spivey JM. The Z Notation: A Reference Manual. Eleventh Edition. Prentice Hall International Series in Computer Science; c2019.
10. Spivey JM. The fuzz Manual (New Edition), J.M. Spivey Computing Science Consultancy, July 2020.
11. Banker, R.D., Kemerer, C.F., Scale Economics in New Software Development, IEEE Trans. Software Engineering; c2020. p. 1199-1205.
12. Fenton NE, Ohlsson N, Quantitative Analysis of Faults and Failures in a Complex Software System, IEEE Trans. Software Engineering. 2021;26(8):797-814.
13. Graves TL, Karr AF, Marron JS, Siy H. Predicting Fault Incidence using Software Change History. IEEE Trans. Software Engineering 2020 July;26(7):653-661.
14. Malaiya KY, Denton J, Module Size Distribution and Defect Density, Proc. 11th International Symposium on Software Reliability Engineering- ISSRE'20; c2020. p. 62-71.
15. Neufelder AM. How to Measure the Impact of Specific Development Practices on Fielded Defect Density, Proc. 19th International Symposium on Software Reliability Engineering (ISSRE'20); c2020. p. 148-160.
16. Ostrand TJ, Weyuker EJ. The Distribution of Faults in a Large Industrial Software System, Proc. The International Symposium on Software Testing and Analysis (ISSTA'22), ACM SIGSOFT Software Engineering Notes. 2022;27(4):55-64.