December 2002

DOES THE iHGOLDILOCKS CONJECTURElS APPLY TO SOFTWARE REUSE? AN EXPLORATORY STUDY USING A DOMAIN-SPECIFIC REUSE MODEL

Derek Nazareth
University of Wisconsin-Milwaukee

Marcus Rothenberger
University of Wisconsin-Milwaukee

DOES THE "GOLDILOCKS CONJECTURE" APPLY TO SOFTWARE REUSE? AN EXPLORATORY STUDY USING A DOMAIN-SPECIFIC REUSE MODEL

Derek L. Nazareth and Marcus A. Rothenberger
University of Wisconsin-Milwaukee
derek@uwm.edu rothenb@uwm.edu

Abstract

The Goldilocks Conjecture represents a speculation that there exists an optimal module size and is based upon behavior observed in software defect prediction models. While some data sets suggest that the Goldilocks Conjecture may apply to defect density prediction, the effect of module size on software reuse effectiveness is not fully understood. Breakeven analysis represents an important aspect of software reuse. This paper examines the breakeven point for repositories of reusable software modules, in relation to both module size and repository size, using a domain-specific model of software reuse. The findings suggest that a variant of the Goldilocks Conjecture may apply, but prescriptions for software reuse practitioners must be made with caution.

Introduction

Software development is generally acknowledged as an expensive and lengthy process, often producing software that is of dubious quality and hard to maintain. The explosive growth in the demand for software, coupled with shortages in the supply of software developers and the stagnant productivity in software development, has only exacerbated the problem. Several different solutions have been proposed as a means of alleviating the situation. Software reuse represents one such proposal. Several benefits have been claimed for software reuse, including reduced development cost and time, improved software quality, increased developer productivity, and improved software maintainability, among others (McClure 1997). These intuitive benefits come at a cost of setting up a repository of reusable modules, and ongoing population and management of the repository. It is expected that long term savings through software reuse will outweigh initial costs of adopting a reuse program. Most studies addressing breakeven points for software reuse tend to focus on post-hoc analyses of cost data for a portfolio of projects. This paper examines the breakeven point for software reuse from a different perspective. Specifically, the effects of module size and repository size are considered in the breakeven analysis. The results indicate a relationship that is non-linear in nature and supports a variant of the Goldilocks Conjecture. While the results are explainable, the implications for software reuse practitioners must be tempered so that spurious recommendations are excluded.

The Goldilocks Conjecture

The concept of optimality has fascinated researchers in the software engineering discipline. Optimality introduces the notion of the best possible performance on a given dimension. Frequently, the notion of optimality has been introduced as a side effect of the application of curve fitting exercises. In the software defect prediction area, there are several studies that examine the relationship between software defects and various attributes of the modules being studied. Polynomial curve fitting techniques have been employed in the creation of defect prediction models (Gaffney 1984; Compton and Withrow, 1990). Many of these models link predicted defects to the size of the module, measured in lines of code. An unintended consequence of using models with polynomial terms is that the model suggests the presence of an optimal size of the module in the context of defect reduction. Gaffney (1984) examined defect densities for assembly language modules. The resulting model predicted the density to be lowest for modules of size 877 lines. In a separate exercise on Ada modules, using a different polynomial model, Compton and Withrow (1990) concluded that the module size that yielded the lowest defect densities was 83 lines. They dubbed this the "Goldilocks Principle", based on the notion that there exists an optimal module size that is "not too big nor too small". Several other

researchers have also experienced similar results when studying defect densities in code. Several possible explanations have been advanced in this context. Basili and Perricone (1984) posit that defects relating to the user interface appear in all modules, and tend to skew the densities for smaller modules. Moller and Paulish (1993) suggest that larger modules may be crafter more carefully, hence initially reducing defect densities as module size increases. Hatton (1994) suggested that human cognitive processing limitations cause the introduction of mode defects for larger modules, causing the defect densities to rise with module size.

These results have some interesting implications for software development. First, it suggests that module size, whether it is measured in terms of lines of code, function points, or any other measure, is a determinant of defect density. This is at odds with the notion of software decomposition, which seeks to break up modules into smaller, more easily crafted, and potentially more reusable modules. Further, it provides a pessimistic outlook for developers engaged in the creation of very small or very large modules, out of necessity. An excellent critique of these results is provided by Fenton and Neil (1999), where they conclude that "the relationship between defects and module size is too complex, in general, to admit to straightforward curve fitting models". Their analysis and results would appear to contradict the notion of the "Goldilocks Conjecture" as a universal relationship between defect density and module size.

Despite the concerns raised by Fenton & Neil (1999) about the relationship between the defect density and module size, there is evidence to suggest that some data sets may support this conjecture (Compton and Withrow 1990; Moller and Paulish 1993; Hatton 1997). It should be borne in mind that the data, however problematic from a quality perspective, merely represents some base underlying facts. As such, it may suggest a model that has some implications for software developers. However, attempting to read beyond these implications in terms of prescription for software development is something that is problematic. Thus, in the context of software defects, developers of very small and very large modules need to be cautioned about the propensity for higher defect densities. Any suggestion that there exists and optimal module size that developers should strive to achieve with a view to reducing defect densities is misguided and potentially harmful.

The Goldilocks Conjecture and Software Reuse

There is no doubt that software reuse can generate savings in development effort. However, it entails some costs in terms of creating and searching through a repository of reusable components. It is expected that the initial phases of software reuse in an organization will generate negative savings, as the costs incurred with setting up the repository will outweigh any savings accrued through reuse. As the repository grows larger, the savings through reuse will start to offset the costs associated with the initial setup, and a breakeven point will be reached. Beyond this, the savings through should continue to outpace costs, as cataloging costs and search costs are expected to be smaller than development costs averted through reuse. This paper focuses on the breakeven point. Of particular interest to us is whether the breakeven point occurs differently if the repository is populated with small modules, or large modules. If the breakeven varies with module size, the Goldilocks Conjecture may also apply. This has implications for the creation and maintenance of software reuse repositories.

A review of the software reuse literature did not yield any useful material on the effect of module size on the extent of software reuse. However, data from software reuse studies have indicated several non-linear relationships between various reuse parameters and overall reuse costs. In a study of 2954 reused modules at NASA, Selby (1988) determined that a concave non-linear relationship existed between modification effort and percentage of code modified as part of the reuse, whereby small modifications generated disproportionately large costs. Gerlich and Denskat (1994) posit that changes to multiple modules in an application will generate a non-linear set of changes to their interfaces. Cost estimation models for software development in the presence of reuse and reengineering also include non-linear drivers (Clark et. al. 1998). The acceptance of non-linearities in software reuse suggests the potential for second or higher order relationships among software reuse parameters. A second order relationship between module size and other software reuse parameters would indicate the presence of the Goldilocks Conjecture.

Relating Software Reuse Breakeven to Module Size

To investigate a possible relationship between the breakeven point for software reuse and module size, this research employs a domain-specific model of systematic software reuse. The motivation for constraining the model to work with a single domain stems from the notion that reuse is expected to be greatest when the repository of reusable modules address a set of related applications from the same domain. Reuse across domains is expected to be limited, and presents a less interesting scenario. The model addresses cost factors and savings relating to systematic software reuse. It is described in (Nazareth and Rothenberger 2002) and is presented in summary form in the Appendix. The model permits economic investigation of various aspects of software reuse. Prior studies with this model have investigated the effect of project size, module size, and repository size on the magnitude of savings through reuse. In this research, we look at the effect of module size on the breakeven point for software

reuse. An earlier study with this model demonstrated a proportionate relationship between project size and savings, indicating that savings as a proportion of total development time are the same for different project sizes, all other conditions being the same (Rothenberger and Nazareth 2002). Therefore, it can be inferred that project size is not an likely to be an issue in this analysis. On the other hand, we have learned that the repository size affects the search cost, as well as the likelihood to find desired components. Further, the module size also affects the leverage of each reuse instance. These observations suggest that both have an impact on the breakeven point.

The experimental conditions employed in this study are as follows. The repository size was varied from 0 to 2000 modules in steps of 20. The average module size was varied from 20 to 200 lines of code in steps of 1. Expected savings from reuse are computed for these conditions, with particular emphasis on when the breakeven occurs. The results are depicted in Figure 1. Note that for very small repositories, no breakeven was attained, indicating repository creation costs outweighed any savings through reuse. The experiment was repeated for different search costs, both lower and higher than that depicted in the figure. Similar trends were observed, with a flatter curve for lower search costs, and steeper curve for larger search costs.

Implications

The figure suggests that a variation of the Goldilocks Principle may apply to software reuse breakeven analysis. For extremely small repositories, it indicates that breakeven does not occur, which is consistent with the notion that initial repository creation costs are not likely to be offset by the limited opportunities for reuse. However, as the repository starts to grow, it suggests that a moderate module size is needed to break even. This is intuitively understandable in that small modules are not likely to generate sufficient savings when reused. As the repository size grows, the average size of the modules required to break even starts to decrease. Once again, this is understandable in the light of greater opportunity for reuse, and hence a smaller need to rely on large modules for savings through reuse. However, as the repository size increases further, the average size of modules required to break even starts to climb. This can be explained in light of increased search costs and modification costs associated with large repositories.

This data has some tangible implications for software reuse. While it confirms that extremely small repositories are not likely to generate any savings, it also indicates that extremely small modules may not generate enough savings for reuse to be economically worthwhile. Moreover, the change in average module size needed to break even should not be interpreted as a basis for declaring a minimum module size of for example 35 lines of code for inclusion in the repository. Instead, it should be used to caution any software reuse managers of the implications of very small repositories and very small modules.

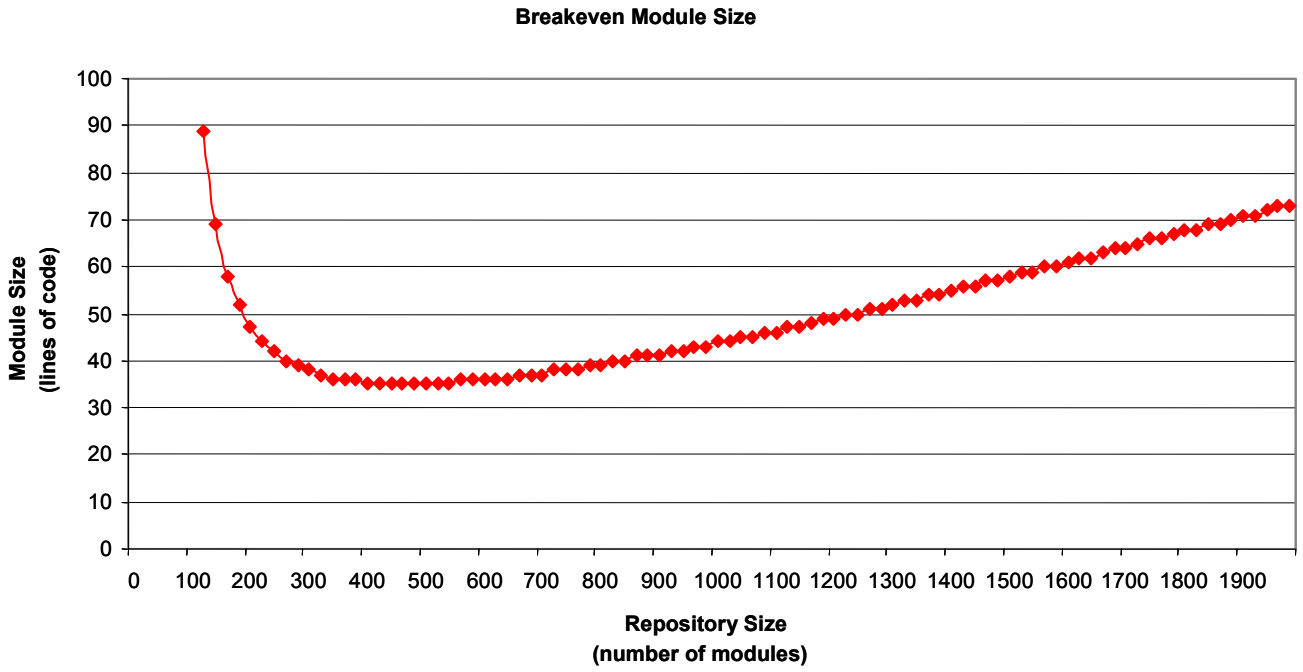

Figure 1. Breakeven Module Size for Software Reuse

Repositories grow over time, as components developed for ongoing projects are included into the reuse library. The results of this study should caution reuse managers about the uncontrolled growth of reuse repositories. This study indicates that adding components to a reuse library will likely reduce the average probability for reuse of an individual module, and increase the search cost, thereby providing only marginal benefit for reuse. Thus, very large repositories run the risk of generating fewer saving if populated by very small modules. This is intuitively understandable, as more components increase the number of reuse opportunities only marginally if the repository already accounts most of the desired functionality in the respective domain; yet, they add to the search cost incurred for each reuse instance.

Conclusions

This study employed a domain-specific model that implements the relationships between costs of a reuse program and savings incurred through reuse to investigate the breakeven points for average module size and repository size. The findings suggest that a variant of the Goldilocks Conjecture may apply, in that the average size of modules required to break even varies with the repository size, and this relationship appears quadratic in nature. It is not the intent of this study to prescribe specific numbers for repository and module size; as such figures are very context dependent. Nevertheless, we have observed an interesting trend that relates breakeven module size and repository size. These results should caution reuse managers about very small repositories, very small modules, and uncontrolled repository growth.

References

Basili, V.R. and Perricone, B.T. "Software Errors and Complexity: An Empirical Investigation", *Communications of the ACM*, 27 (1), pp 42-52, January 1984.

Clark, B., Devnani-Chulani, S. and Boehm, B. "Calibrating the COCOMO II Post-Architecture Model", Proceedings of the 20th International Conference on Software Engineering ICSE-20, 20, Kyoto, Japan, pp. 477-480, April 1998.

Compton, B.T. and Withrow, C. "Prediction and Control of Ada Software Defects", *Journal of Systems and Software*, 12 (3), pp 199-207, July 1990.

Fenton, N.E. and Neil, M. "A Critique of Software Defect Prediction Models", *IEEE Transactions on Software Engineering*, 25 (6), pp 675-689, September/October 1999.

Gaffney, J.R. "Estimating the Number of Faults in Code", *IEEE Transactions on Software Engineering*, SE10 (4), pp. 459-464, July 1984.

Gerlich, R. and Denskat, U., "A Cost Estimation Model for Maintenance and High Reuse," *Proceedings of the European Software Cost Modeling Conference*, Ivrea, Italy, May 1994.

Hatton, L. *C and Safety Related Software Development: Standards, Subsets, Testing, Metrics, Legal Issues*, McGraw Hill, New York, 1994.

Hatton, L. "Re-examining the Fault Density-Component Size Connection", *IEEE Software*, 14 (2), pp 89-98, March/April 1997.

McClure, C. *Software Reuse Techniques*, Prentice Hall, Upper Saddle River, NJ, 1997.

Moller, K.H. and Paulish, D. "An Empirical Investigation of Software Fault Distribution", *Proceedings of the First International Software Metric Symposium*, IEEE CS Press, pp 82-90, 1993.

Nazareth, D.L. and Rothenberger, M.A. "Assessing the Cost-Effectiveness of Software Reuse: A Model for Systematic Reuse", Working Paper, School of Business Administration, University of Wisconsin-Milwaukee, February 2002.

Rothenberger, M.A. and Nazareth, D.L. "A Cost-Benefit Model for Systematic Software Reuse", *Proceedings of the Tenth European Conference on Information Systems (ECIS 2000)*, Gdansk, Poland, June 2002.

Appendix

The model classifies costs associated with software reuse as search costs or publication costs. Search costs address efforts required to locate appropriate modules for reuse in a new software development project. They include query formulation costs, C_Q, and retrieval costs, C_R. Query formulation costs are based on the number of terms to be retrieved, n_q, moderated by the effectiveness of selecting among the query criteria, Q_E. Retrieval costs are based on the number of components to be searched, N, the number of query criteria, n_q, the selectivity among criteria, a_n, and the effectiveness of retrieval, Q_R.

Publication costs, C_P, represent the effort associated with the population of new modules in the repository. This covers the cost of developing new modules, C_D, modification of existing modules for a specific project, C_M, as well as making a module more general for improved reusability, C_G. It also includes the costs of cataloging these modules, C_C. Development costs are based on module size, S, and complexity m_c; modification costs are assessed based on quality of the module, s and its size, S as are the costs for making modules more generic. Cataloging costs are modeled on the basis of the number of cataloging dimensions, n_c, and the fraction of components reused without modification, p.

Benefits from reuse are computed as a simple difference between development costs assuming reuse and development costs assuming no reuse at all. The effect of reuse is based on the nature of the reuse policy (a strict policy will engender less reuse), the proportion of modules reused without modification, and the extent to which a module needs to be modified when it is used in a new project.