SER3093

SOFTWARE DEVELOPMENT RISK MODEL

Applied to Data from Open-Source Mozilla Project

James W. Fawcett, Murat K. Gungor

Abstract — ***Development of large software systems creates many, often thousands, of source code files with complex inter-dependencies. Clusters of mutually dependent files introduce the possibility of a chain of forced consequential changes when a single cluster member file is changed. Our software development risk model shows that density of dependencies within such clusters plays a crucial role in this behavior. We develop a file-rank procedure which orders the entire system's file set by increasing risk. This ranking process should prove to be useful while managing the development of large systems, indicating where attention should be focused to improve Test Risk. We have applied this model to a library from the 1.4.1 release of the open source Mozilla project with interesting results.***

Index Terms — ***Dependency analysis, metrics, open-source, software quality, risk analysis.***

I. INTRODUCTION

Development of large software systems creates many, often thousands, of source code files with complex inter-dependencies. We show, in this paper, that clusters of mutually dependent files introduces the possibility of a chain of forced consequential changes when a single cluster member file is changed, perhaps to repair a latent defect or improve system performance[2]. The model shows that density of dependencies within such clusters plays a crucial role in this behavior. Increasing density leads to increased risk of essentially unending sequences of change, known as thrashing. Our model is derived from a notion of Test Risk, based on the work of Jungmayr[1], combined with a measure of importance, for each file. We develop a file-rank procedure which orders the entire system's file set by increasing risk, the product of importance and test risk, both defined in the paper. This ranking process should prove to be useful while managing the development of large systems, indicating where attention should be focused to improve Test Risk. We have applied this model to a library from the 1.4.1 release of the open source Mozilla project, composed of 598 files of source code, with interesting results.

The results of this paper will, we believe, be useful for any of the disciplines that depend on large complex code bases. Computational Biology, Aerospace Systems, and Medical Imaging Systems, among many others, depend on large software toolkits, analysis systems, and display technology. Because much of the current work in these areas is new research or advanced product development, the codes that support those disciplines are continuously evolving and new software tools appear frequently.

The methods of this paper provide direct support for management of large developing code bases. Not only are weakness discovered, but the model provides direct prescriptive guidance to improve the quality and reduce Test Risk of these systems.

II. DEPENDENCY ANALYSIS

Dependency among software components is necessary to provide services from one component to another; on the other hand, excessive dependencies among components make a system inflexible and fragile. The project becomes difficult for developers to understand, test, maintain and reuse. Using dependency-based and other software metrics, we present a way of diagnosing potential problems of this type in large software systems.

As a project gets larger, dependency among its components gets denser and harder to manage. Therefore it is very important to provide timely feedback to software engineers and project management about the state of the software development project. We focus on file level dependency information, as files are the unit of testing and configuration management. File dependency information can be obtained promptly from source code, using analysis tools, so this information is always available, unlike project documentation, which may be out of date or may not exist.

We've built a tool, DepAnal, which can be used to constantly monitor the state of large software systems and provide guidance about where detailed quality analysis and re-factoring are needed. The tool uses grammar productions for the C and C++ languages that are much simpler than that

Manuscript received April 21, 2005.
Dr. James W. Fawcett is with the Electrical Engineering and Computer Science Department, Syracuse University, Syracuse, NY 13244, USA (phone: 315- 443-3948; e-mail: jfawcett@twcny.rr.com).
Murat K. Gungor is with the Electrical Engineering and Computer Science Department, Syracuse University, Syracuse, NY 13244 USA (phone: 315-443-4003 ; e-mail: mkgungor@ecs.syr.edu).

needed for complete code analysis, and uses the dependency model described below.

Dependency Model [8] - file A depends on file B if:

- A creates and/or uses an instance of a type declared or defined in B
- A is derived from a type declared or defined in B
- A is using the value of a global variable declared and/or defined in B
- A defines a non-constant global variable used by B
- A uses a global function declared or defined in B
- A declares a type or global function defined in B
- A defines a type or global function declared in B
- A uses a template parameter declared in B

We also developed three adjunct tools that provide additional views of the data:

1. Strong Component Analyzer: Builds a dependency graph from the data provided by DepAnal and analyzes its strong components, that is, sets of files that are mutually dependent.
2. Size and Complexity Analyzer: Counts the number of lines of source code in each function and analyzes each function's cyclomatic complexity [6], measured by the number of regions enclosed by the control flow graph of the function. Anal also evaluates the total line count and sum of the complexities of all of the functions in each file.
3. Dependency Viewer: Generates 2D graphical display of components and their dependency relationships.

We also use the finance toolbox of Matlab [3] to solve simultaneous linear equations that result from the risk model described in this paper.

III. SOFTWARE DEVELOPMENT RISK MODEL

In the Figure 1, an arrow shows the dependency relationship among two source files, where each square represents a source file. If the arrow points from file A to file B, then file B provides services to A and A depends on B.

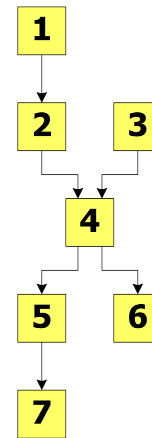
In this example, files 6 and 7 are the most independent files since they do not depend on any other files' services. It is straightforward to test them, at least in terms of these structural relationships. However, this does not imply that these files are unimportant. On the contrary, files 6 and 7 provide services to many files above them, so their importance in this example is high.

To discover the state of software system, we develop a file-rank procedure which orders the entire system's file set by increasing risk, the product of importance and test risk.

This is leads us to define two things:

1. Importance of a file
2. Test risk of a file

In the following section importance and test risks are elaborated.



In this project file 1 has high test risk, due to its dependence on all the other files except file 3, either directly or indirectly. But its importance is low, in that no other files depend upon it for services. The opposite is true of files 6 and 7. Files 2, 3, 4, and 5 are intermediate cases that we will analyze below.

Figure 1: Simple dependency between files

A. *File Importance*

Here we define importance from the perspective of change impact. Importance, I, can be greater than or equal to 1. File 1 has importance 1 ($I_1 = 1$), since no other files depend on file1, it can be changed without worrying about anything other than its internal implementation. If we pick a file which is being used by other files, it will have higher importance, since any change applied to that file may affect the files above it.

$$I_i = 1 + \sum_{AllCallers} \alpha_{ij} I_j$$

Here we use coefficient alpha (α_{ij}), which shows the risk of impact on files j caused by change in file i. Thus, if there is no risk that a change in file i will affect file j, there is no contribution, from that file, to the importance of file i.

The smaller (closer to 1) the importance value for file i is, the better, in terms of impact of modifications to this file on the remaining files in the system.

α_{ij} is impact strength, which indicates the affect on upper level files of changes in called files. If it is certain that a change in file 2 will cause a change in file 1, $\alpha_{21} = 1$, and the importance of file 2 is 1 + $\alpha_{21} = 2$, e.g. the number of files changed when file 2 changes.

If α_{ij} evaluates close to 1, it indicates that upper level files will be affected significantly by changes occurring in lower level files which provide services, so importance will increase rapidly.

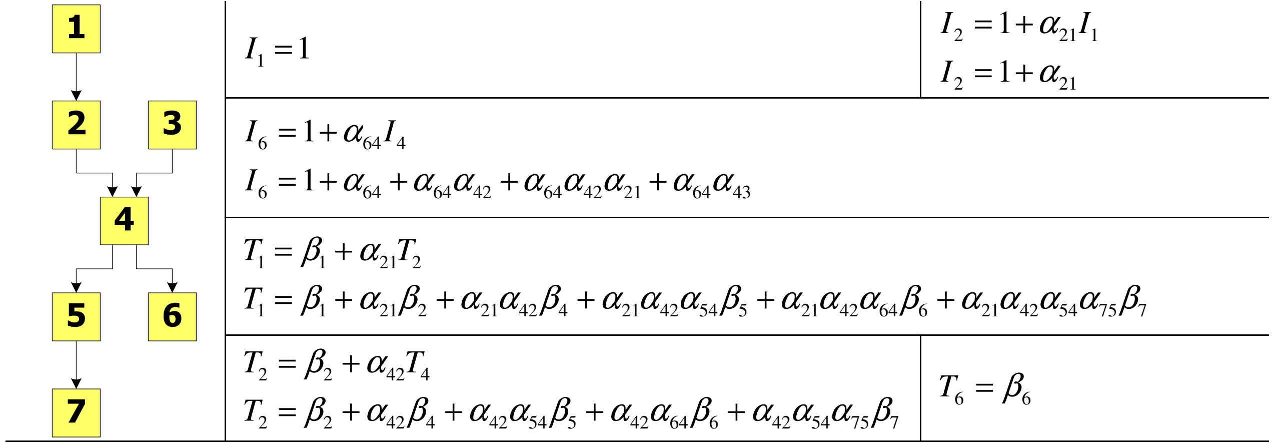


Figure 2: Example of importance and Test Risk of files.

If α_{ij} is close to 0, it indicates upper level files will not be affected much by changes occurring in low level files and the lower level files are not so important.

A. *Test Risk of File, T*

Test Risk of a software file is an important issue in assuring that required functionality is implemented without errors. "*A lack of testability contributes to a higher test and maintenance effort*" [1]. Testing a file that uses services of others is harder than testing a file that performs its required task without depending on other files. In Figure 2, Test Risk of file 7 is the lowest rank. The smaller T (close to 1) is, the more testable the file.

Below, we introduce implementation quality (β), which is described in section B

$$T_n = \beta_n + \sum_{AllCalled} \alpha_{mn} T_m$$

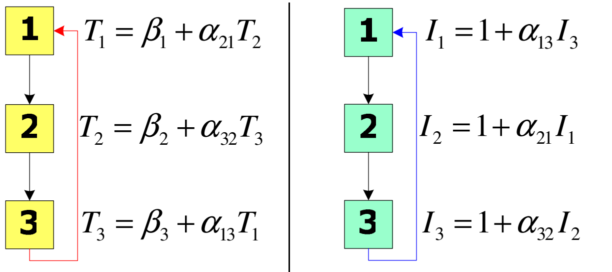


Figure 3: Circular dependency.

Magnitude of Test Risk metric varies according to the depended upon files' internal structure and the project's dependency structure. β_n is the test risk of file n in isolation. T_n is the test risk accounting for retesting necessary when one of the file's dependent files changes and it must change.

B. *Implementation Metric Factor,* β

Test Risk of a file depends not only on its internal implementation quality, but also on the quality of the files that it depends on. For this reason, metric factor, β , of many other files in the project may affect the test risk of any specific file. A number of metrics may be chosen to evaluate β . For this paper we use average lines of code per function and average cyclomatic complexity per function. For our own work we take 50 lines of code and cyclomatic complexity of 10 as upper bounds of desirable values for these metrics. We use these bounds to normalize the metric factor, as follows:

$$\beta_i = 1 + \frac{1}{N}\sqrt{(\frac{m_{1i}}{M_1})^2 + (\frac{m_{2i}}{M_2})^2 + ... + (\frac{m_{Ni}}{M_N})^2}$$

$$\beta_i = 1 + \frac{1}{N}\sqrt{\sum_{j \in (1,N)} (\frac{m_{ji}}{M_j})^2}$$

Lowercase m is the measured metric, uppercase M is boundary value metric.

C. *Case of Circular Dependency*

In the case of circular dependency, each member has the same importance and Test Risk size, since there is a mutual dependency between each file, and any change can affect any other files, as shown in Figure 3.

$$T_1 = \beta_1 + \alpha_{21} T_2$$
$$T_1 = \beta_1 + \alpha_{21}(\beta_2 + \alpha_{32} T_3)$$
$$T_1 = \beta_1 + \alpha_{21}(\beta_2 + \alpha_{32}(\beta_3 + \alpha_{13} T_1))$$
$$T_1 = \frac{\beta_1 + \alpha_{21}\beta_2 + \alpha_{21}\alpha_{32}\beta_3}{1 - \alpha_{21}\alpha_{32}\alpha_{13}}$$

$$I_1 = 1 + \alpha_{13} I_3$$
$$I_1 = 1 + \alpha_{13}(1 + \alpha_{32} I_2)$$
$$I_1 = 1 + \alpha_{13}(1 + \alpha_{32}(1 + \alpha_{21} I_1))$$
$$I_1 = \frac{1 + \alpha_{13} + \alpha_{13}\alpha_{32}}{1 - \alpha_{13}\alpha_{32}\alpha_{21}}$$

In Figure 3, we see the effect of circular dependency over Test Risk and importance. As identified, α_{ij} are always less than 1, dividing importance by $1 - \alpha_{13}\alpha_{32}\alpha_{21}$ or Test Risk by $1 - \alpha_{21}\alpha_{32}\alpha_{13}$ makes Test Risk and importance increase. Thus circular dependency increases Test Risk, since a change in any file may affect every file in the mutual dependency set.

Figure 4 and Figure 5 show the matrix representation of importance and Test Risk for Figure 3.

$$\begin{bmatrix} 1 & 0 & -\alpha_{13} \\ -\alpha_{21} & 1 & 0 \\ 0 & -\alpha_{32} & 1 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Figure 4 – Matrix representation of importance

$$\begin{bmatrix} 1 & -\alpha_{21} & 0 \\ 0 & 1 & -\alpha_{32} \\ -\alpha_{13} & 0 & 1 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

Figure 5 – Matrix representation of Test Risk

When there are more than a single cyclic path there is a critical value for α_{ij} at which the solution for importance and Test Risk becomes singular, e.g., the risk becomes unbounded. This indicates that a change made on a component with unbounded risk is likely to cause an unending sequence of changes[1].

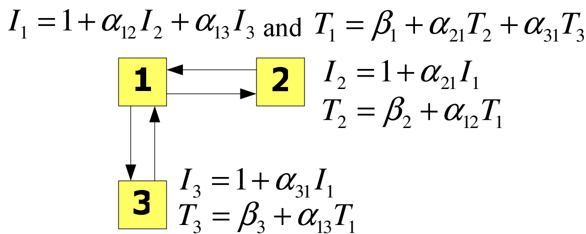


Figure 6 – Three mutually depended files.

In Figure 6, if for all i, j, α_{ij} are greater than 0.7071, behavior becomes undefined, as the change sequence becomes unbounded.

[1] Essentially, our risk model is a Markov process that becomes unstable at the critical value for α_{ij} .

$$I_1 = \frac{1 + \alpha_{12} + \alpha_{13}}{1 - \alpha_{12}\alpha_{21} - \alpha_{13}\alpha_{31}}$$

and

$$T_1 = \frac{\beta_1 + \alpha_{21}\beta_2 + \alpha_{31}\beta_3}{1 - \alpha_{21}\alpha_{12} - \alpha_{31}\alpha_{13}}$$

It can be clearly seen in Figure 6 that circular dependency increases the software system's Test Risk and file importance. Importance increases since a change in any given file affects all files in the mutually dependent set, including possibly itself. A few more simple cases with increasing numbers of paths show that as density of dependency paths increase the critical value for α_{ij} decreases.

D. *Risk of a File, R*

Risk factor is calculated by product of importance and Test Risk metrics.

$$R_i = I_i x T_i$$

A file with high Importance and high Test Risk will have a high risk, while a file with low importance but the same high Test Risk will have lower Risk Factor.

We develop a file-rank procedure which orders the entire system's file set by increasing risk, R_i , the product of Importance and Test Risk. This ranking process should prove to be useful while managing the development of large systems, indicating where attention should be focused to improve Test Risk.

Risk factor provides feedback about individual files and also provides insight about the global state of a software project. For instance, if developer needs to test a file, risk factor will give an idea how much time to allocate for that task. Ranking files by Test Risk shows project management where to focus effort to reduce overall risk by redesigning and re-factoring high risk files.

IV. EMPIRICAL STUDY OF RISK MODEL ON MOZILLA LIBRARY, GKGFX

We downloaded version 1.4.1 of the Mozilla Win32 configuration [4] [5]. This included the entire build, which makes many executables and libraries. We were able to build all the libraries and executables in about a week's effort, using the information provided on www.mozilla.org.

We built some simple parsers to find all the files included in a specific build, based on compiler output. This included all common code and header files.

The information provided on the Mozilla web site was very well prepared, easy to digest, considering the size of this

large project, and straightforward to use. We chose this project because of the quality of its tools and the fact that it has a very large code base.

We applied our risk model to Mozilla GKGFX library and it gave us important insights about potential problem files, on which attention should be focused. This information obtained without diving into implementation details, which is very important for the software project's testers, developers, and managers.

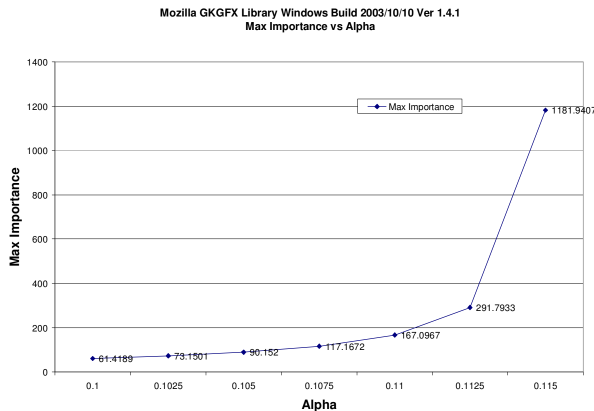


Figure 7 – Max Importance vs. Alpha (α) value for Mozilla GKGFX Library Version 1.4.1.

First, we explored the variation of maximum importance with $\alpha_{ij} = \alpha$, making the simplifying assumption that it is constant for all files. Essentially we are treating α as the average probability of a consequential change in a depending file when we change the depended file. Thus, these results will be qualitatively useful, but not numerically precise. We see, from the plot in Figure 7, that Importance grows without bound above α =0.115. This indicates that changes are very likely to propagate throughout the system since one might expect the value of α to be of the order of 0.1.

Next, we calculated Risk factor values using average cyclomatic complexity[2] (AvgCC) and Fan-out[3] values for each file in GKGFX when calculating β; upper limits were 10 for AvgCC and 5 for Fan-out. We took these values since it becomes harder to manage a file which uses several other files' services, accordingly, it is hard to understand and test a file with high complexity functions.

Figure 8 shows the risk rate of all the files in Mozilla library, GKGFX, in increasing order. Note that about %10 of the files have most of the Risk in the library code. Interestingly, the file with the highest risk is part of the second largest component (with 56 files in Figure 9). This shows that the high risk files are not guaranteed to be part of largest strong component.

[2] AvgCC = Sum of CC of each functions in a file divided by number of functions in that file.
[3] Fan-out is a number of depended files whose services are employed by a file.

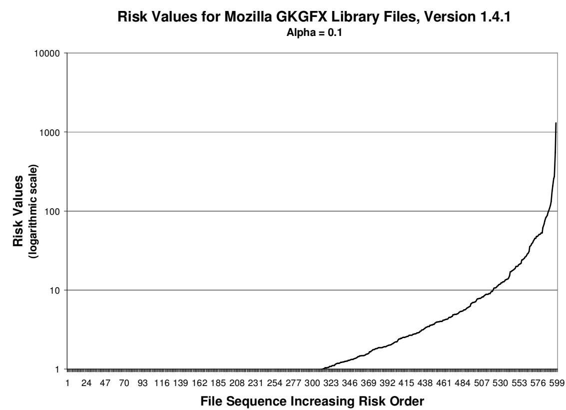
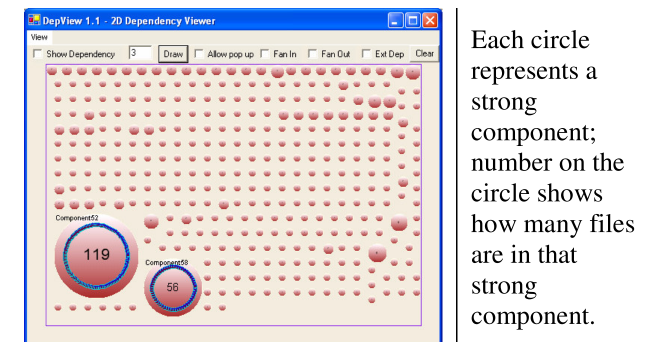


Figure 8 – Risk values for files in GKGFX Library



Each circle represents a strong component; number on the circle shows how many files are in that strong component.

Figure 9 – Components of GKGFX Library

Not surprisingly, all the files with high risk are members of strong components [9]. This also proves that risk analysis is providing dependable information.

V. CONCLUSIONS

In this paper we present a new software development Risk Model and have shown that the model can be used to predict problem areas, as concentrations of high risk files. The model predicts that, as the density of dependency relations increases in strong components of the dependency graph, Risk factor grows and becomes unbounded at critical densities. We've applied the model to a library from a real open-source project where the model predicted that most of the development risk is in about 10% of the library files.

REFERENCES

[1] Stefan Jungmayr, "Identifying Test-Critical Dependencies", Proceedings of the International Conference on Software Maintenance (ICSM'02), IEEE, 2001.

[2] M.M. Lehman and L.A. Belady, Program Evolution: Processes of Software Change. Academic Press, 1985.

[3] http://www.mathworks.com/ Documentation > Financial Toolbox > Solving Simultaneous Linear Equations

[4] Mozilla the Configurator, http://webtools.mozilla.org/build/config.cgi
[5] Mozilla on Microsoft Windows 32-bit Platforms, www.mozilla.org/build/win32.html
[6] Tom McCabe, “A complexity measure”, IEEE Transactions on Software Engineering, 2(4), pp. 308-320, 1976
[7] Katsuro Inoue, Reishi Yokomori, Hikaru Fajiware, Tetsuo Yamamoto, Makoto Matsushita, Shinji Kusumoto, “Component Rank: Relative Significance Rank for Software Component Search”, 25th International Conference on Software Engineering, 2003.
[8] James Fawcett, Murat Gungor, Arun Iyer, "Analyzing static structure of large software systems", SERP'05 The 2005 International Conference on Software Engineering Research and Practice, Nevada, USA 2005.
[9] J. Lakos. Large-scale C++ software design. Addison-Wesley, 1996.

James W. Fawcett (M’61–LM’04) received his PhD degree in Electrical Engineering from Syracuse University. His research interests include software complexity and developing infrastructure to re-engineer software reuse processes and make accessible, for reuse, not only code, but also documentation and test products.

Murat K. Gungor received his BS degree in industrial engineering from Sakarya University in Turkey, and received his MS degree in computer science from Syracuse University. Currently (2005) he is continuing his PhD study at Syracuse University. His research interests include static software analysis, software change and using software metrics to understand and improve static structure of large software systems.