TÜBİTAK

Turk J Elec Eng & Comp Sci, Vol.20, No.1, 2012, © TÜBİTAK
doi:10.3906/elk-1007-566

New change impact factor estimation in software development

Murat Kahraman GÜNGÖR[1], **Ersin ELBAŞI**[1,*], **James Walter FAWCETT**[2]
[1] *The Scientific and Technological Research Council of Turkey (TÜBİTAK), Tunus Caddesi No: 80, Kavaklıdere, Ankara-TURKEY*
e-mails: murat.gungor@tubitak.gov.tr, ersin.elbasi@tubitak.gov.tr
[2] *Electrical Engineering and Computer Science Department, L.C. Smith College of Engineering and Computer Science, Syracuse University, Syracuse, NY 13244, USA*
e-mail: jfawcett@twcny.rr.com

Received: 13.07.2010

Abstract

Change in software is always an essential part of software development and maintenance. Estimating a proposed change's effect on the later phases of the development helps project managers and developers with decision-making and predicting future progress. During development, on some occasions, speedy solutions are necessary to meet project schedules. Such quick changes may lead to major quality flaws in the long term, even though they solve local problems in the short term. Controlled management of change is achieved by being able to estimate the impact of changes. In this paper, we propose a new change impact factor estimation and present the design of an experiment to measure these effects, describe its application, and show the measured results of the change impact.

Key Words: *Change impact factor, software development, product risk model*

1. Introduction

In this research, we report on measurements of the impact of change in one file on other files in a small design project called DepAnal. DepAnal is one of the tools that were monitored throughout its evolution for this paper. We will describe this measurement as change impact factor α_{ij} and define it as:

$$\alpha_{ij} = \frac{\sum \text{Changes in file } j \text{ due to a change in file } i}{\sum \text{Changes in file } i} \tag{1}$$

Thus, the change impact factor (CIF) is the relative frequency of required consequential changes in files in the project. In an earlier research effort [1-4], a product risk model was developed that uses change impact factors for every dependency relationship between files in a project, but it could supply only rough estimates

*Corresponding author: The Scientific and Technological Research Council of Turkey (TÜBİTAK), Tunus Caddesi No: 80, Kavaklıdere, Ankara-TURKEY

for the values of these parameters. The goals of the present effort were to measure the CIFs as functions of time for a real project, and also to develop a measurement process that can be applied to other projects, as well. In this way, a more accurate assessment of risk is obtained, in real time, as a project unfolds.

We present the design of this experiment, describe its application, and show the measured results of the change impact factors. These results help one to estimate the propagation [5] of changes and calculate the magnitude of change, the CIF, for a project. The results of the study will improve the accuracy of the risk analysis model [1] calculation by using systematically measured change impact factors derived from an annotated change history. Consequently, all of this information provides help to developers and project managers to find the parts of their product that are at risk. Not only that, but it also guides them to make effective decisions with regards to implementing new changes and scheduling work activities.

The results of this paper will 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.

2. Related works

Lee [6] defines the objective of change impact analysis this way:

A major goal of impact analysis is to identify the software work products impacted by proposed changes. Evaluating software change impacts requires identifying what will be impacted by a change and relies on the "impact assessment" to determine quantitatively what the impact represents.

Her dissertation [6] considered the impact of change on types, global functions, and global data, such as how many classes are going to be affected by a change. Similar analyses are also found in [7].

In this study, we are interested in a coarser level of impact analysis, that of file-to-file change impact. Our choice is motivated by the conventional process of managing projects by files. Files are the unit of configuration management and analysis. Our risk model is based on file dependencies, calculated from the same kinds of static relationships used in [4,6-8], e.g. type, function, and global data. However, change impacts are empirically determined by carefully monitoring and recording original and consequential changes made to files during development.

3. Change impact factor and risk model

The granularity of change impact factor in this research work is focused on software source files. We are interested in determining the degree of interconnectedness between source files to be able to estimate consequences of a change. The degree of interconnectedness is represented by α.

α_{ij} is the likelihood of a consequential change in file j when a change occurs in file i, as shown in Figure 1. The arrows show directions of change causality. Given any 2 files, i and j, there are 2 different alpha values between them. One is α_{ij} and the other is α_{ji}. $\alpha_{ij} = 0$ is the lower bound, implying that changes in file i are not going to affect file j. $\alpha_{ij} = 1$ is the upper bound, indicating that any change in file i is going to affect file j. α_{ij} and α_{ji} are inherently 2 different alpha values, as will be demonstrated in the following section.

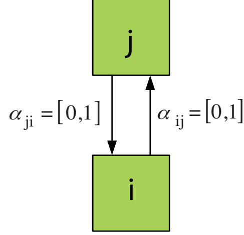

Figure 1. Alpha-value representations.

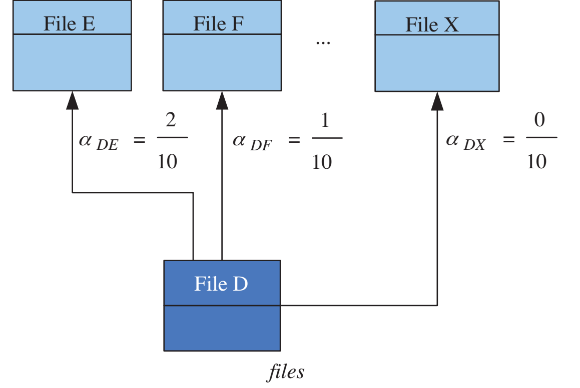

Figure 2. Alpha values betwen file D and dependent files.

An alpha value is the ratio of the number of consequential changes made to a file to the total number of changes in a source (of change) file. The total number of changes is the sum of the original and consequential changes. In Figure 2, the alpha calculation is carried out by dividing the number of consequential changes that occurred to files E, F... X via file D by the total number of changes in file D.

In Figure 2, a sample alpha-value calculation is illustrated. File D is providing services to files E, F, and X. There are a total of 10 changes occurring in file D, and 2 of them required file E to change. The calculation is as follows.

$$\alpha_{DE} = \frac{2}{10} = \frac{\text{Consequential changes to } E \text{ caused by changes in } D}{\text{Total changes in } D} \tag{2}$$

The product risk model ranks files according to internal implementation metrics and external interactions with other files in the project [1]. The risk factor, R, is the product of importance and testability for file i, $R_i = I_i x T_i$. Both the importance and testability of file i, I_i and T_i, respectively, use alpha values during their calculation, as shown in the formulas below.

Table 1. File testability.

Importance of file i	Testability of file n
$I_i = 1 + \sum_{AllCallers} \alpha_{ij} I_j$	$T_n = \beta_n + \sum_{AllCalled} \alpha_{mn} T_m$

Importance, I, can be greater than or equal to 1. If we pick a file that is being used by other files, it will have higher importance, since any change applied to that file may affect the files above it. α_{ij} is the impact strength, which indicates the effect on upper-level files of changes in the called files. The 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, the 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 [1].

In Figure 3, file 1 has high test risk due to its dependence on all of the other files except file 3, either directly or indirectly. However, 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.

In Figure 4, we show the risk, importance, and testability values for each file of DepAnal [2], the tool that analyzes static dependencies between files. The benefit of the product risk factor [1] is that it provides

both feedback about individual files and insight about the global state of a software project. For example, in Figure 4, we see the risk contributions of each file to the project and see immediately the files that pose high and low risk for the project.

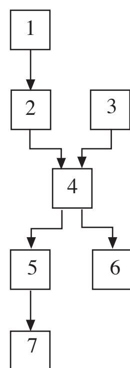

Figure 3. Simple dependency between files.

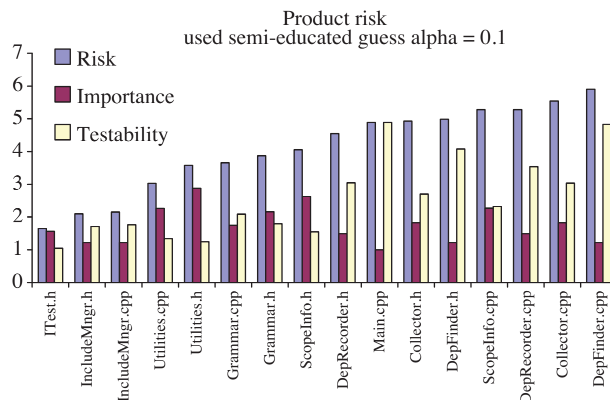

Figure 4. Risk chart of new design of DepAnal [2].

Before testing a file, its product risk factor provides an idea of how much effort to allocate for that task; it also shows where to focus effort to reduce overall risk by redesigning and refactoring high-risk files.

Risk factor is calculated as the product of importance and test risk metrics.

$$R_i = I_i \times T_i \tag{3}$$

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 a lower risk factor. We developed a file-rank procedure that orders the entire system's file set by increasing risk,

R_i, the product of the 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 [1].

In earlier research [1], alpha values were modeled as a single constant, 0.1; this was just a semieducated guess. One of the aims of the present study was to achieve experiment-based alpha values to support the risk model. In addition, this will enable us to compare the results obtained through the constant alpha with empirically obtained results to observe whether differences in the alpha value radically affect the ranking of files by risk values. In our results, we saw that over 62% of the files either stayed in place or moved at most 2 places, as compared with the file risk order obtained by individually calculated alpha values.

4. Experiment design to determine alpha (α)

We designed an experiment to empirically determine alpha values and observe their changes over time. There are 2 essential points in this experiment design. The first is to determine what is meant by a "change"; the second is to have a software project that is large enough to be a reasonable yardstick with which to measure other systems, but small enough to monitor implementation from start to end. Thus, we obtained a sufficient number of sample data points to correctly represent more general software systems.

By change, we mean a modification/addition/removal of code for any purpose (feature addition, bug removal, commenting, and cosmetic changes) to a file. In addition, making a group of cosmetic changes at once

is a change. Each file has its own change history and each change is part of a daily file release in our project. Until the first release, the changes made to a file will be counted as one change. First release will be counted as first change. Consequential change is a specific change, required to accommodate a previous change in another file. All other changes are by default original changes, indicating that they are not initiated by another change.

One exception with consequential change is that, in the same version, if a change requires more than one consequential change to a particular file, only the first change to the particular file should be recorded as consequential, and the rest will be recorded as original changes.

As an example, consider the case where a single change in file A causes one or more changes to file B. In Figure 5, changes labeled as C2, C3, and C4 are due to change C1; however, only C2 is recorded as a consequential change and the rest are original changes.

Only direct changes due to a source change, not the transitive closure, are counted as consequential changes. Note that a consequential change may cause another consequential change.

Figure 6 shows sample dependency structure and changes. Here, C1 is an original change; C2 through C5 would not happen if C1 did not occur. Nevertheless, we record that C3 and C5 were caused due to C2, not due to C1. C4 is recorded as a consequential change, too; however, C6 and C7 are recorded as original changes.

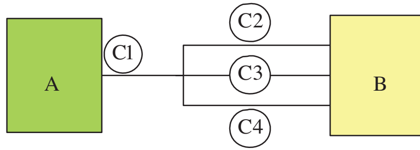

Figure 5. A change driving many changes.

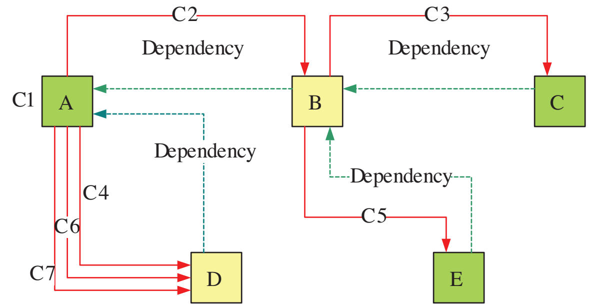

Figure 6. Sample change flow and dependency between files.

Experiment details:

- A file release can exhibit one or more changes.
- After each successful compilation, all of the tests should be exercised to make sure there is no breakage. If the breakage requires a fix in other files, this is recorded as consequential change(s).
- Each change is recorded in a maintenance page (comment section within each file) with the date and change number. For example, in Ver. 2.1.a, 2.1 represents the version number of that file and "a" indicates the first change in this version. This is also done for implementation files (.cpp, etc). Since our granularity is file-level, we do not record changes for modules, but always record them for individual files.
- If a new function's declaration and definition are added to different files (header and implementation), we record each as a change in the maintenance history page of the corresponding files. To be consistent, we always accept declaration as original change and definition as consequential change.
- During a fix, or a new feature addition, if several changes are required in the same function, this will be counted as one change, provided that previously developed functionality remains intact.

- During a modification or a fix, if a new global function is created, there will be at least 2 changes; 1 is the fix/modification and the other is the new function. Nevertheless, it is not a consequential change, since both reside in the same file.
- If a new class-member function is created, there will be a total of 3 changes; declaration and definition of the new function will both be consequential changes, declaration will be a consequential change of the fix, and definition will be a consequential change of the declaration.
- Addition of a new member variable is a consequential change of declaration required by implementation.
- Adding/removing an existing source file to/from a system is a change.
- Removing an already added file is not a change, provided that it is not supplying any services to others. If it does supply services, it will cause several external changes; therefore, file removal will be a change.
- While adding/removing an existing source file to/from a system, all are original changes. We do not differentiate such that A.h is an original change and A.cpp is a consequential change.

5. Empirical study process description

This section covers some practical details of the experiment. The sample data for this experiment came from a reimplementation of our C/C++ file-level dependency analyzer [1,2]. The analyzer's first external release has 7796 lines of evolved code, and 5580 of these are code within functions. Implementation took 3 months, and 503 changes were recorded.

Table 2. Information regarding the experimental project.

Statistical information on the analyzer	
Total code lines	22,553
Evolved code lines	7796
Total evolved function lines	5580
Total cyclomatic complexity in evolved code	812
Time to first external release (months)	3
Number of changes recorded	503

Each change is recorded in a maintenance page for each file in which the change occurred. A change record contains the following information:

- Date
- Change number, qualified with internal release number
- Brief information regarding the nature of the change
- Whether it is a consequential change or not

We also created a change logger application, shown in Figure 7, to keep data in an organized fashion in order to query later. The change logger carries extra information regarding each file, shown in Table 3. These extra data are used for exploring correlations between metrics (structural or internal) and changes. This will be a topic of future research.

Table 3. Information in database regarding a file in which change occurred.

ChangeLogTable : Table

Field Name	Data Type
ChangeNo	AutoNumber
FileName	Text
ChangeType	Text
Comments	Memo
CausedBy	Number
FanIn	Text
FanOut	Text
MaxCC	Text
TotalCC	Text
AvgCC	Text
MaxFunSize	Text
AvgFuncSize	Text
NumOfFunc	Text
SCSize	Text

- Change type
- Number of dependent files (FanIn)
- Number of dependents on file (FanOut)
- Cyclomatic complexity
- Maximum and average function size
- Total lines of code
- Etc.

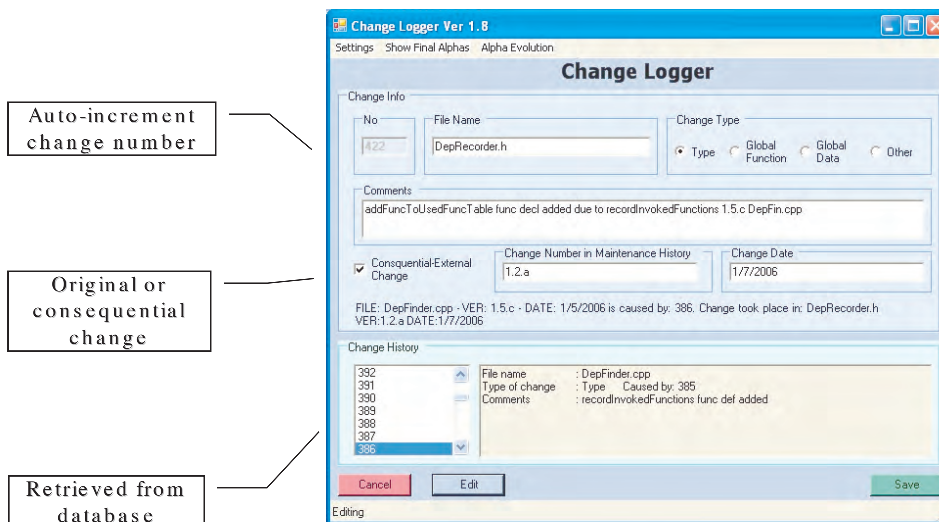

Figure 7. Screen shot of change logger.

Once a developer implements a change, he has to record it both in the database by using the change logger (Figure 7) and in the maintenance page, before working on other parts of the software. To record a change, the following information is needed: filename where change occurred, brief textual explanation regarding the change, change number, type, and date. If it is consequential change, the developer has to select the file that caused the change.

Alpha-value evaluation can be monitored for any period of time during the development. Alpha values between any 2 files can be extracted to see their interaction over time.

Figure 8 shows the alpha calculator, which can extract alpha values between any times during project development. In addition, it generates matrix files to be used for product risk calculation.

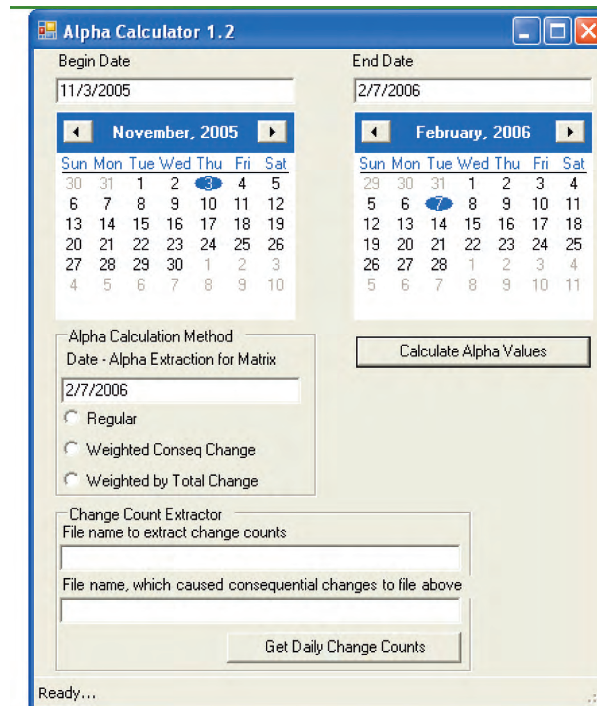

Figure 8. Alpha-value calculator.

6. Experimental results

This research provides several graphical outputs. One graphical output type is evaluation of an alpha (CIF) value chart for each individual file. Another is evaluation of the project's alpha value chart throughout development. CIF charts for a file have 2 forms; the first shows the number of consequential changes that occurred in file A due to changes in other files. We show this CIF value as α_{XA}. X is the file causing consequential changes to file A. The other form of chart is the number of consequential changes caused by file A to other files. We show this CIF value as α_{AX}. Changes are cumulative change counts over some certain time interval.

The file's alpha evaluation chart discloses information about how likely this file is to be affected by the changes in other files, or how likely it is that changes in this file will affect other files. The chart below (Figure 9) shows the alpha value of file Collector.cpp, such as $\alpha_{\text{Grammar.cpp,Collector.cpp}}$. We read $\alpha_{\text{Grammar.cpp,Collector.cpp}}$ as the fact that a change occurred in Grammar.cpp and how likely it will be that change is required in Collector.cpp. The file's alpha evaluation charts below do not disclose dependency information.

When we mention consequential change, generally, the scenario is as follows. If file A is using the services of file B, a change in file B causes A to change. However, in some cases, it can be just the opposite, such that while file A is using a feature of file B, it can encounter a bug and request file B to change. Another example is that file A can request a new feature addition from file B. In the former case, consequential change is just the reverse direction of the dependency, but in the latter cases, it is the same direction of dependency.

In this chart, we first see a sharp rise in the alpha value ($\alpha_{\text{Collector.h,Collector.cpp}}$), and then it becomes stable. In most cases, the header and implementation files have a higher alpha value compared to other files. This is tolerable, since they are intended to accomplish assigned tasks together. Until design ideas settle down, frequent changes are normal.

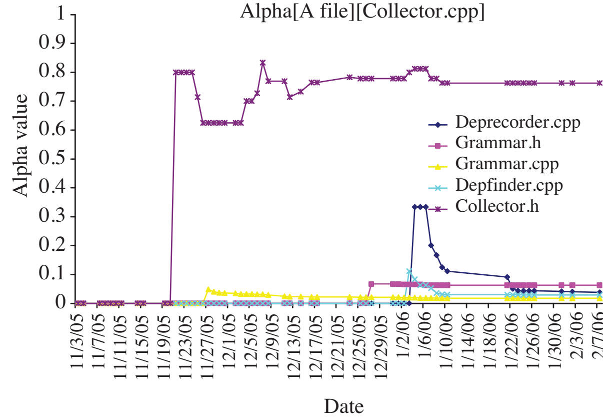

Figure 9. Alpha value evaluation of Collector.cpp throughout the first release.

Most changes between modules (header file and implementation files) are due to function signature change, adding/renaming/removing member data, or function. All of these changes are legitimate and frequent changes between modules at early stages of development since design ideas are not settled yet.

$$\alpha_{A.Collector.cpp} = \frac{\sum \text{Consequential changes in Collector.cpp due to } A}{\sum \text{Changes in } A} \quad (4)$$

To be consistent while recording changes, we accept that definitions always depend on declaration. As we see, $\alpha_{\text{Collector.h,Collector.cpp}}$ is quite high, implying that almost any change in Collector.h affects Collector.cpp. This is because any member function addition starts with its declaration and then its definition. This means that all of the function definitions are consequential changes as a result of the high alpha value between the header and implementation file.

The lower the denominator is, the higher the fraction. If there are not many internal changes recorded in file A or if all of the internal change causes the subject file to change, this can cause the α value to be high. Lower alpha values indicate files' level of independence from external changes. Therefore, a lower alpha value is better. When we see an equal rate of reduction in the alpha value, it indicates that changes occurring to a causing file are not causing consequential changes to the subject file.

When there is an increase in the alpha value, it indicates that consequential changes are occurring to the subject file. When the alpha value decreases or remains the same, it means that there is no significant change taking place in it. Charts also disclose information regarding a file's creation or inclusion time in the project. By looking at the timeline in Figure 10, it is seen that this file is created in the early stages of the project.

Figure 10 shows changes in $\alpha_{\text{Collector.h,Collector.cpp}}$ during the time frame of 1 month, ignoring changes that occurred before. In addition, the continuous line shows the alpha-value change of $\alpha_{\text{Collector.h,Collector.cpp}}$ by taking the change history into account for comparison. This allows us to monitor the alpha value over some time interval. Both lines are close to each other in Figure 10. However, if there was no change after November 23, we would expect 2 distant lines. The beginning value difference is due to not considering past changes.

Figures 9 and 11 both show alpha values for Collector.cpp. In spite of the fact that both figures show the alpha value for the same file, grammar.cpp does not appear in Figure 11. This is because no consequential change occurred in Collector.cpp due to that file during the time period covered. Moreover, alpha values in Figure 11 are different than values in Figure 9 on the same days. This is due to ignoring the past changes.

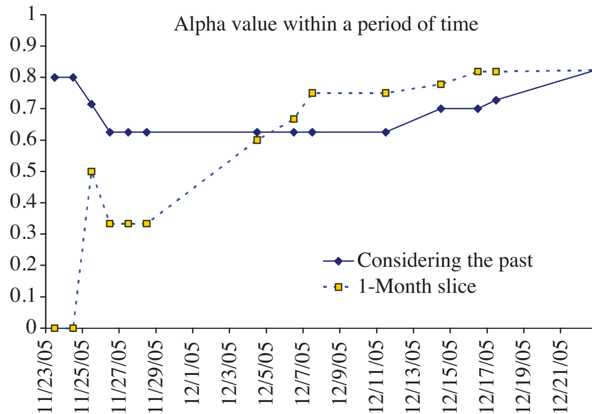

Figure 10. Alpha value evaluation in 1-month period between Collector.h and .cpp.

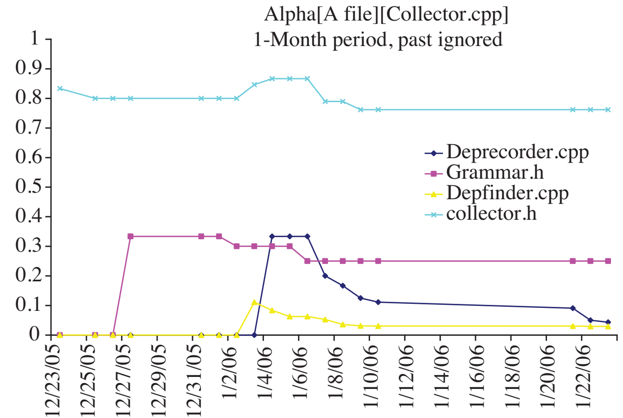

Figure 11. Alpha values evaluation for 1-month period.

The sliding-window time frame is useful for monitoring the evolution of alpha values over certain time periods. Another benefit is to eliminate the effect of history and see the real alpha values during a certain period of time.

$$\alpha_{Collector.cpp\ \ A} = \frac{\sum \text{Consequential changes count in } A \text{ due to Collector.cpp } A}{\sum \text{Changes in Collector.cpp}} \tag{5}$$

In Figure 12, we see how changes in Collector.cpp spark other files to change. Between the header and implementation file, there is a relatively higher alpha value than the others. One important thing affecting the alpha value is the number of changes that occurred in Collector.cpp. After January 10, no changes occurred to Collector.cpp since the lines are parallel to the axis.

Figures 12 and 13 both show alpha values of Collector.cpp. Similar to in the charts above, the alpha values are different, since change history is totally disregarded in Figure 13. This indicates that the changes recorded in Collector.cpp until the beginning date of the time frame were ignored. If, during the time period covered, the ratio of original to consequential changes is small, it could result in the surfacing of higher alpha values.

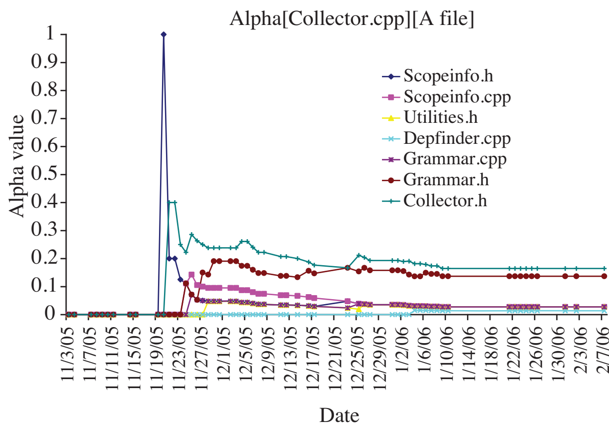

Figure 12. Alpha-value evaluation of Collector.cpp throughout the first release.

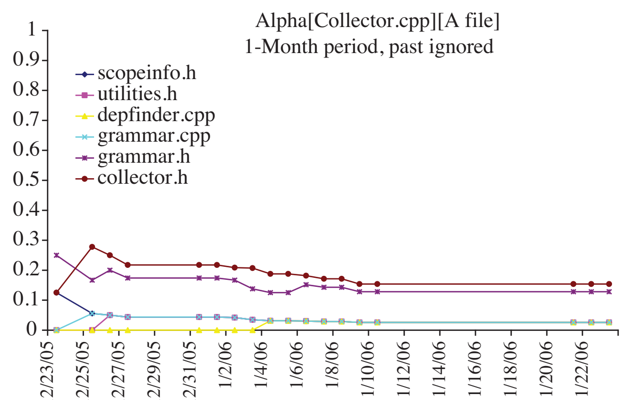

Figure 13. Alpha-value evaluation for 1-month period.

If there is no change in alpha values (after January 10 in Figure 13), there is no change occurring to the causing file. This could be an indication that the file has met its planned functionalities and is fulfilling its requirements, or it could be the project manager's decision not to make changes until a particular project release to meet the schedule and budget.

The pruned average alpha value is a single alpha value, which represents the overall project, calculated by summing non-zero consecutive change counts that occurred in a time unit (day) and averaging by the number of changes that occurred on that day, as described by the formula below. All of the changes are cumulative. In Eq. (6), m is the number of files in a project, and n_i is the number of files to which file i causes consecutive change. t is the time during the development of a project.

$$\alpha^{t}_{\text{Pruned average}} = \frac{\sum\limits_{File\,i}^{m} \sum\limits_{F\ \text{ile}\,j}^{n_i} \text{Consecutive change}^{t}_{ij}}{\sum\limits^{m} \text{Change}_i} \ \text{if Consecuitive change}^{t}_{ij} \neq 0 \tag{6}$$

In Figures 14 and 15, the evolution of the pruned average alpha value over some time interval is shown. Figure 14 covers the time frame starting from the beginning of the project up to the time of the project's first release. At the beginning of the project, the pruned average alpha value is low, since the files are trying to place initial internal features. Figure 15 covers a 1-month slice of the development time, ignoring the number of changes at the beginning. During the time period covered, files started to use evolved files services. Due to use of services of other files, naturally frequent testing occurred. This testing uncovered bugs and increased the need for additional functionality. As a result, consequential changes occurred, and therefore a higher alpha value is observed in Figure 15.

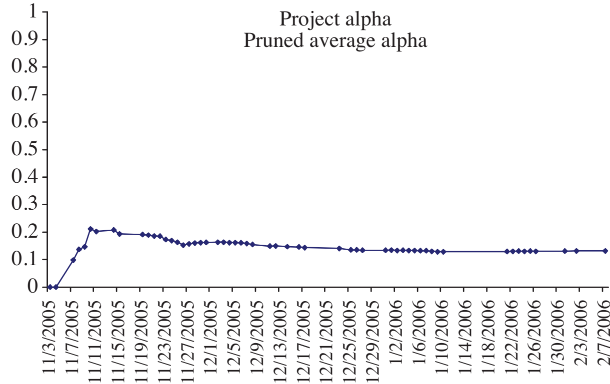

Figure 14. Pruned average alpha-value evaluation throughout the first release.

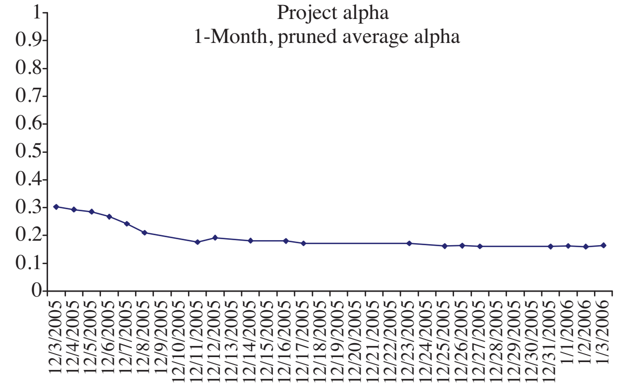

Figure 15. Pruned average alpha-value evaluation for 1-month period.

7. Risk analysis with measured alpha values

Figures below show the product risk [1] of files in the experimental project (DepAnal) calculated using measured alpha values. Figure 14 also shows product risk, but the alpha value is a semieducated guess there. Risk values in Figure 16 were obtained by individually calculated alpha values, meaning that each α_{ij} value (change impact value) used was measured using change history.

We know that the risk values obtained with individually calculated alpha values are the most precise ones. On the other hand, if single alpha values were used for overall analysis, we wondered how closely the single alpha would represent real risk values. For that purpose, we calculated the risk values by using the pruned average alpha, which is a single alpha measured using change history, as in Eq. (6). Figure 17 shows the risk values calculated with the measured project alpha value.

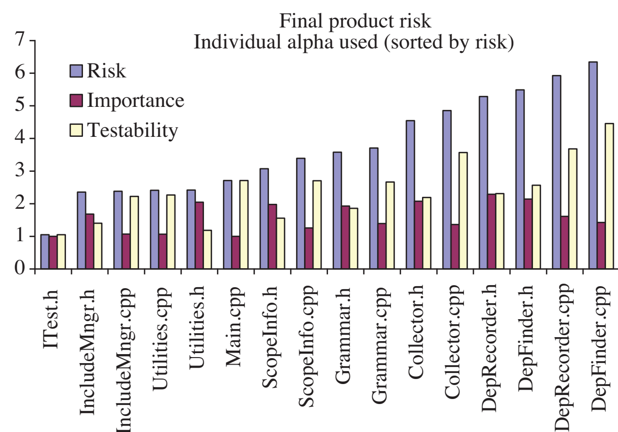

Figure 16. Product risk with individually calculated alpha.

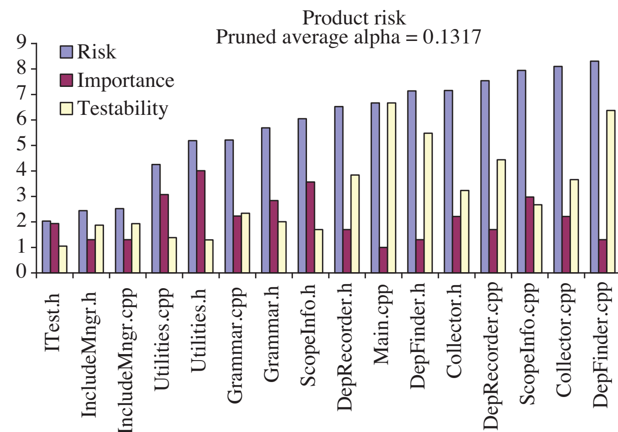

Figure 17. Product risk with pruned average alpha (single).

As we see in Table 4 and Figure 17, more than 62% of the files stayed in the same order or moved at most 2 places, as in Figure 16. Therefore, the order difference between using a single measured pruned average alpha and individual alphas can be disregarded for this experiment.

Table 4. Change in risk ordering of files calculated by measured pruned average alpha and guessed alpha with regard to risk calculated by measured individual alphas.

Ordering change with regard to individually calculated alpha				
	Semiguessed alpha		Calculated pruned average alpha	
Order shange	Count	Percentage	Count	Percentage
Same place	6	37,50%	7	43,75%
1 space moved	2	12,50%	1	6,25%
2 space moved	2	12,50%	3	18,75%
3 space moved	2	12,50%	1	6,25%
4 space moved	3	18,75%	3	18,75%
5 space moved	0		1	6,25%
6 space moved	1	6,25%	0	

The effort spent for obtaining individual alpha calculation is not negligible. If alpha calculation is automated, this will be of great help for obtaining precise risk values, which is an interesting future research area.

Figure 18 shows the comparison of product risk results. As we see, the risk value calculated with semieducated alpha guesses and pruned average alphas have mostly the same slope, but different values. Nevertheless, risk with individually calculated alpha values shows the most accurate values; interestingly, risk with semieducated alpha guesses is closer to them for this experiment.

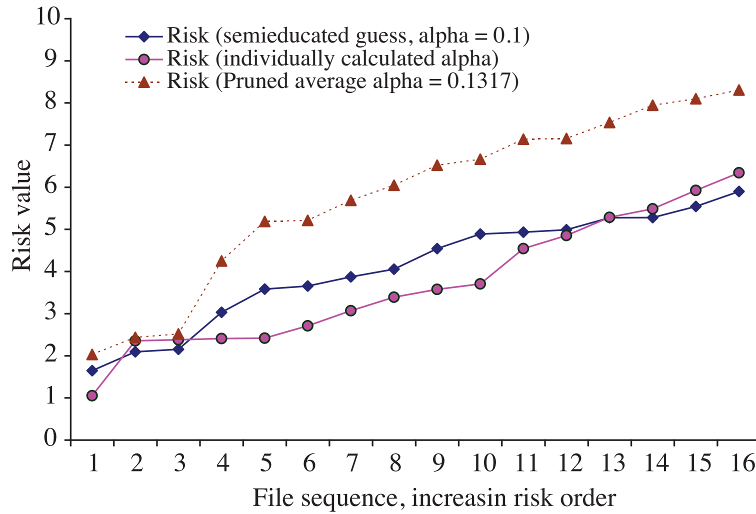

Figure 18. Comparison of outcome of product risk with alpha variance.

8. Conclusion

Change in software is an essential part of software development and maintenance. Estimating a proposed change's effect on the later phases of the development helps project managers and developers with decision-making and predicting future progress. During development, on some occasions, speedy solutions are necessary to meet project schedules. Such quick changes may lead to major quality flaws in the long term even though they solve local problems in the short term. For example, software can acquire a tendency toward consequential changes, or unintended dependencies can arise. Management of change is achieved by being able to estimate the impact of changes; this study serves mainly to support that need. Calculated risk profiles disclose each file's vulnerability to external changes, as well as the project's overall vulnerability. Software managers can use these charts to monitor and control the change process. Understanding impacts of a change is one of the methods for guarding against software quality degradation.

In the change impact-based approach, the first step consists of mapping the source code change to a set of atomic changes. This method use classes, methods, fields, and their relationships as the atomic units of change [10]. Calibrating CIF parameter values for a project from the change history is applicable to any software development project. These quantitative measurements are superior to semieducated guesses. High change impact values are not a desirable property of a file or a project. If a file is inclined to change due to external changes, this increases the effort required for implementing changes. As a result, it increases the bug fix-time and the new feature implementation time. Knowing the system's sensitivity to change and estimating the effect of a change enables controlled and well-planned change activity.

Experiments show that using change history enables us to:

- Understand the degree of connectedness between the source files,
- Provide controlled change activity,
- Monitor software quality,
- Understand the evolution of CIF over a project's lifetime, and

- Determine the quality of software via CIF; a high CIF indicates low quality or an immature software project. Alpha values help to predict a change based on the average of how many other changes it initiates. Consequently, this helps during:
    - Decision making,
    - Effort estimation, and
    - Project scheduling.

References

[1] J.W. Fawcett, M.K. Gungor, “Software development risk model. Applied to data from open-source Mozilla project”, 2005 International Conference on Software Engineering Research and Practice, Las Vegas, NV, USA, 2005.

[2] J.W. Fawcett, M.K. Gungor, A.V. Iyer, “Analyzing static structure of large software systems based on data from open-source Mozilla project”, 2005 International Conference on Software Engineering Research and Practice, Las Vegas, NV, USA, 2005.

[3] S.L. Pfleeger, S.A. Bohner, “A framework for software maintenance metrics”, IEEE Transactions on Software Engineering, pp. 320-327, 1990.

[4] S. Bohner, R. Arnold, Software Change Impact Analysis, Los Alamitos, CA, USA, IEEE Computer Society Press, 1996.

[5] S. Barros, T. Bodhuin, A. Escudie, J.P. Queille, J.F. Voidrot, “Supporting impact analysis: a semi-automated technique and associated tool”, Proceedings of the Conference on Software Maintenance, pp. 42-51, 1995.

[6] M. Lee. Change Impact Analysis of Object-Oriented Software. PhD Dissertation, George Mason University, 1999.

[7] M. Lee, A.J. Offutt, R.T. Alexander, “Algorithmic analysis of the impacts of changes to object-oriented software”, TOOLS-34 ’00, 2000.

[8] J. Law. G. Rothermel, “Incremental dynamic impact analysis for evolving software systems”, Proceedings of the International Symposium on Software Reliability Engineering, pp. 430-441, 2003

[9] S. Jungmayr, “Identifying test-critical dependencies”, Proceedings of the International Conference on Software Maintenance, IEEE, pp. 404-413, 2001.

[10] B.G. Ryder, F. Tip, “Change impact analysis for object-oriented programs”, ACM Workshop on Program Analysis for Software Tools and Engineering, Vol. 1, pp. 46-53, 2001.