Using the Ripple Effect to Measure Software Quality

Haider Bilal and Sue Black

Centre for Systems and Software Engineering,
Faculty of Business, Computing and Information Management
London South Bank University
{bilalhz,blackse}@lsbu.ac.uk

Abstract

Software metrics provide us with information regarding the quality of software. The ripple effect metric shows what impact changes to software will have on the rest of the system. It can be used during software maintenance to keep the system at an optimum level of quality, avoiding degradation of the system or during development to ensure that the quality of the system is maintained throughout the development process. The computation of ripple effect is based on the effect that a change to a single variable will have on the rest of a program; it determines the scope of the change and provides a measure of the program's complexity. The original algorithm used to compute ripple effect has been reformulated mainly to provide clarity in the operations involved. The reformulation involved some approximation that was shown not to affect the measures produced. The reformulated, approximated algorithm has been implemented as the software tool REST (Ripple Effect and Stability Tool). This paper describes the ripple effect measure and shows how it can be used to maintain system quality throughout the software lifecycle.

1.0 Introduction

Software plays an important role in our lives. We want products that affect our lives to have quality attributes. We need good quality software and in order to determine the quality of software we need metrics to measure that quality. A key point here is that the quality of a product may change over time and software is no exception. So, if we determine the quality attributes of the software, we can also have more precise, predictable and repeatable control over the software development process and product. If the software engineer knows what to do, then we can measure software more easily. In order to make wise choices about certain software development practices, we should be able to measure the effectiveness of those practices. However, there is a problem, termed “dysfunction” by Austin in 1996 [1] which is commonly faced by businesses and other organizations:

Measurements are made of the things which are easy to measure, leading to strong pressure for improved performance along these dimensions. At the same time, no measurements are made of the things harder to measure, and no pressure applied there. However, for any given dimension of performance, "ease of measurement" does not necessarily correlate well with "criticality to business results". In fact, the correlation usually goes the other way. Measurement efforts therefore tend to have adverse effects, because people respond to the differential pressure by slacking off on the aspects not measured, even if they are critical to business results.

An excellent example of this is software quality. Productivity, roughly defined as the number of features (or worse, lines of code) delivered per unit time, is all too easy to measure. Quality is much harder to measure, because even at its simplest it consists of several distinct dimensions, e.g. customer satisfaction with the product delivered, programming defects detected during the development process or after deployment of the product, and various "abilities" such as maintainability. Austin's model - a purely qualitative one - does a lot to explain the classic paradox of software development: the more pressure you put on a team to have it deliver faster, the later the project will actually ship as a result of poor quality. Conversely, Austin's model also explains the virtuous circle of agile software development: the more effort you invest in quality, in all its aspects - even the less easily measured ones - the less your software projects will actually cost and the faster they will go. Thus quality
assurance: building quality into software is cheaper and produces more efficient software than quality control: looking at quality after the software has been produced.

1.1 Software Measurement

To improve quality during development, we need models of the development process, and within the process we need to select and deploy specific methods and approaches and employ proper tools and technologies. We need measures of the characteristics and quality parameters of the development process and its stages. We need metrics and quality models to help ensure that the development process is under control to meet the quality objective of the product. What is measured is improved. Data and measurements are the most basic prerequisites for the improvement and maturity of any scientific or engineering discipline. Yet, in the discipline of software engineering, this area is perhaps one that has many critical problems and one that needs concerted effort for improvement.

The use of measurement, metrics, and models in software development assumes the availability of good data. In fact, the poor quality of data is a large obstacle in quality improvement. In general, data gathered during the formal machine testing phases are more accurate than those collected at the front end, such as requirements analysis and design reviews. Some companies do not even collect data at the front end of the development process, and for them in-process quality management means only monitoring data during the formal testing phases. To enhance data accuracy, a good tracking system for the entire development process must be in place, and the system must address the data validation issue. Measurements for software projects, therefore, should be well thought out before being used. Metrics that are arbitrarily established could be harmful to the quality improvement effort of a company, and there are numerous examples of this in industry. Each metric used should be subjected to an examination of the basic principles of measurement scale, the operational definition, and validity and reliability issues should be well thought out [2].

1.2 Software Evolution and Maintenance

Software evolution refers to the on-going enhancements of existing software systems, involving both development and maintenance. Software maintenance has been recognized as the most costly and difficult phase in the software life cycle [3]. Over the life of a software system, the software maintenance effort has been estimated to be frequently more than 50% of the total life cycle cost. This maintenance cost shows no sign of declining [4]. Unlike many other types of products, software products are intended to be adaptable. Even though software neither deteriorates nor changes with age if its media are well presented, software maintenance is an expensive process where an existing program is modified for a variety of reasons, including correcting errors, adapting to different data or processing environments, enhancing to add functionality, and altering to improve efficiency [5].

For programs with many interacting modules, modifying and then revalidating a program is complex: analysis, testing, and debugging may be required for each module individually and for the interactions among modules. The problem is further compounded because the maintainers are rarely the authors of the code and usually lack a complete understanding of the program. Even worse, maintainers often do not have access to specifications or design documents – just the code. As software ages and evolves, the task of maintaining it becomes more complex and more expensive [6]. Decades of research on maintenance activities in the procedural software have produced several conclusions. Among them is the recommendation that a reduction in maintenance cost could be achieved by a more controlled design process, and by more rigorous testing of potential problem areas early in the life cycle [7]. For maintenance work to be effective, it is vital to control the input to the process – the procedure by which change requests are notified and managed in the first place. The software maintenance process can only be optimized if precise and unambiguous information is available about the potential ripple effects of a change on an existing system [8].

This section has been an introduction to software quality, software measurement and software maintenance in the context of the goals of this paper. The rest of this paper is structured as follows: section two describes a particular type of software measurement: impact analysis, and a particular type of impact analysis: ripple effect. Section three describes how the ideas contained within impact analysis and ripple effect could be used at each stage of the software lifecycle to ensure that the quality of the software is enhanced and maintained. Section four outlines our conclusions and details further work that needs to be carried out to implement the ideas contained in this paper.
2.0 Impact Analysis and Ripple Effect

As the software industry has matured, we have shifted our resources from being devoted to developing new software systems to making modifications in evolving software systems: software maintenance. A major problem for developers in a changing environment is that seemingly small changes can ripple throughout the system to cause major unintended impacts elsewhere. As such, software developers need mechanisms to understand how a change to a software system will impact the rest of the system.

2.1 Change Impact Analysis

Of the total maintenance cost, 40% lies in rework (i.e. change) of software architecture, component interaction, procedures/methods, and variables. Experience shows that making software changes without understanding their effects can lead to poor effort estimates, delays in release schedules, degraded software design, unreliable software products, and the premature retirement of the software system [9]. The two most expensive activities in software maintenance are the understanding of problems or other expressed needs for change, in relation with the understanding of the maintained software system, and the mastering of all the ripple effects of a proposed change. A seemingly small change can ripple throughout the system to have major unintended impacts elsewhere. As a result, software developers need mechanisms to understand how a change to a software system will impact the rest of the system. This process is called change impact analysis [10].

Change impact analysis improves the accuracy of resource estimates, provides better scheduling, and can reduce the amount of corrective maintenance, because fewer errors will be introduced. One example is the Year 2000 (Y2K) problem. In the past, memory and disk spaces were precious resources, and some old systems used two digits to express the date. As these software systems evolved, legacy software was not extended to address the date requirement of the new century [11]. Organizations attempting to address the Y2K problems discovered that impact analysis was essential to its solution. Without effective analysis to identify ripple effects of changing date variables, a great deal of time is needed to manually examine source code to identify date variables, change them, and test them, only to find that other variables that use the date are also impacted. Moreover, other software objects may also need to be examined and modified to be consistent with the Y2K changes. Those changes could in return, impact the code that has been changed and tested. Now, this software has to be changed and re-tested again. Estimates vary as to the cost of correcting the Year 2000 Problem in industry, but are probably all in the region of billions of dollars [11].

2.1.1 Benefits of Impact Analysis

Experience has taught us that comprehensive up-front analysis of requirements during software development pays high dividends by reducing the risk of costly rework and the potential for errors in planning estimates. The same concept appears to hold true for software change impact analysis. By identifying potential impacts before making a change, we greatly reduced the risks of embarking on a costly change because the cost of unexpected problems generally increases with the lateness of their discovery [11]. Impact analysis information can be used for planning changes, making changes, accommodating certain types of software changes, and tracing through the effects of changes. It makes the potential effects of changes visible before the changes are implemented to make it easier to perform changes more accurately and identifies the consequences or ripple effects of proposed software changes during development and maintenance [11]. There is often more than one change that can solve the same problem or satisfy the same requirement. Assessing the complete impact of each change is often necessary to be able to choose which change to apply. There are also, sometimes, external constraints that must be taken into account when designing the change, such as packages to be interfaced with or parts of the system that must not be impacted. Impact analysis helps the maintenance team identify software work products impacted by software changes. Such analysis not only permits evaluation of the consequences of planned changes; it also allows trade-offs between suggested software change approaches to be considered [11].

Impact analysis can be used as a measure of the cost of a change. The more the change causes other changes, the higher the cost is. Carrying out this analysis before a change is made allows an assessment of the cost of the change and helps management choose tradeoffs between alternative changes. It allows managers and engineers to evaluate the appropriateness of a proposed modification. If a change that is proposed has the possibility of impacting large, disjoint sections of a program, the change might need...
to be re-examined to determine whether a safer change is possible. Ripple effect, which is described in the next section, is a type of impact analysis.

2.2 Ripple Effect

The term “ripple effect” was first used in a paper by Haney [12] to describe the way that a change in one module would necessitate a change in any other module. He used a technique called ‘module connection analysis’ which applied matrix algebra to estimate the total number of changes needed to stabilise a system. Myers [13] used matrices to quantify matrix independence: a complete dependence matrix was formulated describing dependencies between modules within a system and then used to predict the stability of the system. Soong [14] used the joint probability of connection, a measure which looked at the probability that certain elements within a system were connected to other elements within the same system, to produce a program stability measure. All of the aforementioned methods use matrices to measure the probability of a change to a variable or module affecting another variable or module. Yau and Collofello’s ripple effect [15] uses ideas from Haney, Myers and Soong’s work, but their ripple effect measure is not a measure of probability.

When Yau and Collofello first proposed their ripple effect analysis technique in 1978 [15] they saw it as a complexity measure which could be used during software maintenance to evaluate and compare various program modifications to source code. Computation of ripple effect involved using error flow analysis where all program variable definitions involved in an initial modification represented primary error sources from which inconsistency could propagate to other program areas. Propagation continued until no new error sources were created. An algorithm for computing design stability was presented in [16] which facilitated computation of stability based solely on design information. It was proposed that a design stability measure would be more useful than previous stability measures because it could be used at a much earlier stage in the software lifecycle, before any code was produced, thus potentially saving time and money.

The computation of ripple effect was reformulated in 2001 to make the calculation more explicit. The reformulation revealed how the algorithm’s structure can be broken down into separate parts thus providing clarity and enhancing the understanding of its structure. To facilitate the software implementation of the new algorithm an approximation was made, greatly simplifying the calculation that is important for automatic ripple effect computation. The explicit details and validation of the approximated calculation are described in [17].

![Diagram](image.png)

**FIGURE 1. Intramodule and intermodule change propagation**
2.3 Computation of Ripple Effect

The computation of ripple effect is based on the effect that a change to a single variable will have on the rest of a program. Given the three lines of code contained in Module m1 shown in Figure 3: a change to the value of b in (1) will affect the value of a in (1), which will propagate to a in (2). In (2) a will affect d which will then propagate to d in (3). Propagation of change from one line of code to another within a module is called intramodule change propagation.

Change can also propagate from one module to another. When d is returned in (3) the change will propagate from module m1 to module m2, propagation from one module to another module is called intermodule change propagation. All of this information concerning intramodule and intermodule change propagation is held in the following matrices:

- Matrix V contains the starting points for propagation within the module
- Matrix X contains the propagation information within the module
- Matrix Z contains the propagation between modules

Another matrix:

- Matrix C contains a complexity factor (McCabe’s Cyclomatic complexity [18].

The information held in these matrices is then used to compute ripple effect. Broadly speaking ripple effect for a module is an aggregate of the intramodule and intermodule propagation for that module. Ripple effect for a program can be computed as an aggregate of the ripple effect of all modules within the program. A tool to compute ripple effect for C source code has been produced: REST (Ripple Effect and Stability Tool) [19]; a C++ parser for the tool is currently being implemented to allow computation of ripple effect for object-oriented programs.

3.0 Using Ripple Effect throughout the software development lifecycle

Software is supposed to change. So, why does the software community struggle with problems of software maintenance and its requisite change? Much of the concern has more to do with the complexity and sheer size of current applications than it has to do with change. As we develop large software systems (now in the millions of lines of code) incorporating more features and newer technology, the need for new impact analysis technology has emerged [20, 21]. Changing requirements are endemic to software [22]; many researchers have written about software changes and their consequences [23, 24, 25]. Final requirements seldom exist for software systems since they are continually being augmented to accommodate changes in user expectations, operational environment, and the like. Therefore, many software systems are never really complete until their function in the organization becomes obsolete.

Development cycles for large software systems can be lengthy. Continuously changing requirements represent a key management challenge for these efforts. Similarly, software change cycles for large, complex systems can be slow. Without requisite change impact analysis and management mechanisms, software changes during maintenance can have unpredictable consequences that often delay their implementation [26]. Impact analysis is integral to software release planning and the software maintenance process [27]. Impact analysis supports release planning by identifying software life-cycle objects (SLOs) that are likely to change for each software change proposed in a set of change requests. Knowing these SLOs enables project managers to make more precise effort and cost estimates of the software changes [21]. These estimates support planning for software releases by providing necessary information to determine what can be reasonably included in a software release over a given period of time [28].

Basic software change activities can be summarized as: understanding software with respect to the change, implementing the change within the existing system, and retesting the newly modified system. Each of these activities has some element of impact determination. To understand the software with respect to the change, we must ascertain parts of the system that will be affected by the change and
examine them for possible further impacts. While implementing the change within the existing system, we need to be aware of ripple effects caused by the change and record them so that nothing is overlooked. Once the change has been designed and implemented, we need to find existing test cases for regression testing and test cases that may need to be re-examined for redesign based on new requirements [26].

### 3.0.1 Software Maintenance Models

Over the years, several software maintenance models have been proposed, often to emphasize particular aspects of software maintenance. Among these models, there are common activities. The following is a brief summary of software maintenance models reported in the literature.

Boehm’s maintenance model [29] consists of three major phases: understanding the software, modifying the software, and revalidating the software. The Martin-McClure model is similar, [30], consisting of program understanding, program modification, and program revalidation. Parikh [31] has formulated a description of maintenance that emphasizes the identification of objectives before understanding the software, modifying the code, and validating the modified program. Sharpley’s model [32] has a different focus; it highlights the corrective maintenance activities through problem verification, problem diagnosis, reprogramming, and baseline reverification. Osborne’s model of software maintenance [33] concentrates on managing the maintenance activities and determining appropriate measurements applied for visibility, but not into impacts of changes. The Yau and Patkow models are useful in evaluating the effects of change on the system to be maintained. Yau [34] focuses on software stability through analysis of the ripple effect of software changes. A distinctive feature of this model is the post-change impact analysis provided by the evaluation of ripple effect. This model of software maintenance involves: 1) determining the maintenance objective, 2) understanding the program, 3) generating a maintenance change proposal, 4) accounting for the ripple effect, and 5) regression testing the program.

Since the consequences of software change are detected within an existing system, the impact analysis process is modeled in the context of software maintenance. Software changes are invoked by change requests generated by users. The change request and the current software system are key inputs to the Software Change Process. The key output of this process is a new software system which is turned over to end-users for use. During the system’s operation more potential enhancements and repairs are detected. These provide input for the next iteration of changes. The primary activities of the software change process involve understanding the software change, specifying and designing the software change, implementing the change and retesting the affected software.

The classical representation of the software development process is a sequence of, at least, 3 main activities: specification, design, coding. Each of these activities has developed its own techniques. The end of each activity must be marked by the preparation of a document which describes the current state of development of the system. Therefore, we can view the software development model as a process of producing a sequence of documents, each one derived from the previous, and describing from the predecessor the same software system at a greater level of implementation. The rest of this section is a description of how impact analysis and ripple effect can be used throughout the software lifecycle [26].

### 3.1 Specification / Requirements

The goal of this first phase of software development is a complete and understandable requirements document and the traceability of all requirements from their source to the software requirements document and then through design, implementation and test. The main impacts are that the wrong software will be developed, that the software will not be developed in time or that the requirements will not be testable.

It is generally accepted that poorly written or rapidly changing requirements are a principle source of project failure. In addition, the later in the life cycle changes are made to requirements, the more resources needed to implement them. Late requirement changes may also cause a ripple effect, causing additional changes in associated areas. During this phase the software engineer works with the systems engineer to verify the requirements and to demonstrate what needs to be done. Early involvement of the software engineer ensures that that the work that needs to be done gets implemented properly. Given an
understanding of the changes, the software engineer then identifies the areas where changes need to be made and can move to the next phase.

In this phase, all module modifications will be clarified. All related, approved, changes selected will need to be documented in the specification document, since the design of these modifications will require an examination of their side effects. All affected modules will be considered and their properties kept consistent. The integration and testing need to be taken into account and planned. By using ripple effect analysis on the proposed changes, we can provide information on and highlight all the directly affected modules. Once all the affected modules identified, we can then review our resources and schedules. All side effects that will be encountered during the impact analysis of the proposed changes will have to be reported as a consequence of the change. Other indirectly affected modules or unpredicted results will have to be taken into account. Test plans based on the ripple effect analysis will be elaborated at this stage as well.

**Understanding Software Change and Determine Impact** is a key activity that evaluates the proposed change request to clarify its requirements and determine possible effects of the change. This activity determines if the change can be made without perturbing the rest of the software and collects change history data to support future analysis. Principle inputs to this activity are the Current System, the initial change request and known ripple effect. They are used to produce a clarified change request, current impacts, traceability, impact/scope and software stability information. These outputs support management and planning aspects of the change process as well as actual software change activities.

**Identify Change Impacts** is a central activity that supports most of the software change process activities by determining the current known software change impacts. The clarify change request activity details the software change using traditional requirements elicitation techniques, requirements analysis and the like. The change request is the principle vehicle for communication the software change information. The activity is guided by software structure, existing requirements and design approach information produced by the review software documentation activity. The identify change impacts activity supports most of the other software change process activities by determining the current known software change impacts. It examines the current system work products, the change request and previously known ripple effect to determine the traceability of the change requirements to existing SLOs, the set of current impacts and an impact/scope summary of the change.

**Determine Software Stability** examines the current impact associated with the change request and the past history of the changes to assess the stability of the software. If small changes result in large efforts then the software is considered unstable. Software that has low stability is not responsive to changes and requires a great deal of effort to maintain.

**Examine Requirements Traceability** assesses traceability of requirements. This activity assumes that traceability information has been predefined during development and is kept current by updates during maintenance. Traceability relationships between SLOs are analysed in light of software change requirements.

### 3.2 Design

The design process translates requirements into a representation of the software that could be assessed for quality before coding begins. The design, also, is documented and becomes a part of the software configuration baseline. However, the products of the design phase are even more difficult to measure. Design products lack the application of standard formats and a standard method for developing the design. This makes design metrics difficult to identify and to collect.

The design of the software is critical to all future stages of development. If the design is very complex, the coding will require more experienced programmers, more meticulous attention to detail, and more time to code and test. Unfortunately, due to the lack of standard design formats, metrics for this phase are often ignored. In this phase, all software modules of the system are documented, and those modules that are shown, by the ripple effect analysis, to need changing, or new ones need adding must be specified. Once the design change is confirmed, the software engineer implements the changes and reviews the changes using his/her modified design checklist.
**Specify and Design Software Change** takes the clarified change request and generates requirements and design specifications. Design is a creative process where the software maintainer models the system with respect to the software change and specifies the requirements and design accordingly. System’s requirements and design specifications are modified with respect to the clarified change request to produce new software requirements and design specifications. This activity also produces measurement information to help manage the changing system.

**Design Program Changes** takes the architecture changes and the current design to produce a new program design specification based on the new requirements specification. The attention to maintenance quality criteria is important here to ensure that the new design does not become too complex or convoluted in the process of making multiple changes to the system design. Design complexity, modularity and documentation are measured and produced as outputs for inputs to the manage software change activity. Ripple effect used during the design process will also ensure that the change is made completely and accurately.

### 3.3 Implementation

The primary goal of a software development project is to develop code and documentation that will meet the project requirements. Two of the main criteria in the implementation phase are that the software must be both maintainable and reusable. Ripple effect analysis can help with this as it is a process which can identify the effects of software changes, coupling across modules may increase and make it more likely that a change in one area will affect another area.

**Implement Software Change** generates proposed changes to the current system from previously produces change requirements and design specifications. This iterative activity involves updating source code according to requirements and design specifications and unit testing to see if they perform correctly. Popular techniques for identifying software change impacts are: data flow analysis, control flow analysis, program slicing and dependency analysis.

The **Determine Modules to be changed** activity examines the current system software modules with respect to the new changes requirement and design specifications to determine the modules that will have to be changed. This activity is guided by the program level impacts that are identified during the change process.

The **Identify Statement Level Changes** activity prepares the way for applying the program changes. This is an important part of the iterative process where the code is analysed, changed, tested and related changes discovered for analysis.

**Apply Program Level Changes** is the activity where the program is transformed. The software change is programmed into the system using module and statement information gathered from the previous activities. This activity produces modified programs to be tested and a measure of adaptability.

The **Test Modified Software Unit** activity addresses the unit test cases necessary to determine if the software operates according to its specification. This entails using unit test cases produced during previous releases of the software and those constructed for the software changes. This activity also accounts for the ripple effect to analyze the propagation of changes to other modules. This will serve as an effectiveness check for the original impact analysis.

### 3.4 Testing

Once the code has been generated and completed, unit testing, formal testing, integration and acceptance testing begins. During formal testing, all software modules are integrated into a cohesive whole and a series of system integration and validation tests are conducted. This is done to find both errors remaining for unit testing and errors which result from unanticipated interactions between subsystems and components. The goals for effective testing are to locate and repair faults in the software, to identify error-prone software and to complete testing on schedule so that the software will operate as well as needed when put into operation.

During the testing phase, as errors are identified, they may result in changes to the code. The impacted code that was identified earlier in the development phase can now be linked to these errors or changes.
This linkage is important because changes to code can lead to additional problems and ripple effect errors. Highly impacted code with multiple changes before release indicates that reengineering of the code should be considered. Also, regression testing must be implemented: the process of testing changes, to make sure that the modified code behaves correctly and that modifications have not affected existing functionalities. Unlike development testing, it uses existing test suites and tests only the components that might be affected by modifications.

Test Affected Software is the final activity before delivery of the modified software. The modifications are tested to assure that they meet new requirements and the overall system is subject to regression testing to verify that it meets existing ones.

The Generate Test Cases for New Functionality activity takes updated software along with requirements and design specification information to construct new test cases. As testing progresses, incomplete test cases may be discovered and updates to the test suite may be necessary. Changes to tests for one part of the software may change tests to other parts of the software test suite.

3.5 Maintenance

The software changes of the future require that we account for what we do to the software at each stage of development. Therefore at each stage of development and maintenance we must ensure that the software system documentation is up to date. This perspective on software evolution saves considerable time in making future changes to the software.

The attributes fed back to management are aspects of overall software maintainability. The maintainability measures give indicators of the overall quality of the resulting product. Low software maintainability results in difficult and costly software maintenance activities. By monitoring product quality with each change, this process model can be used to control maintenance quality. Software change impact analysis in the process provides the necessary visibility to make effective changes to software systems.

3.7 Summary

Impact analysis plays a key role in the software change process. For the Design Software Change activity, requirements and design impacts are determined. For the Implement Software Change activity, program level impacts are identified. For the Test Affected Software activity, test impacts identify the appropriate tests. The ripple effect used during each of these activities augment the set of known impacts and furnishes a more refined impact analysis.

4.0 Conclusions and Further Work

Because software now plays a very important role in our lives we need to ensure that our software products are of good quality. Using ripple effect and impact analysis as part of a software measurement program can give useful feedback which can then be used to improve future iterations of the product. Previous work in this field has concentrated on measuring ripple effect during implementation and design phases of the software development lifecycle. This paper has described how impact analysis and particularly ripple effect can be used throughout the software lifecycle to ensure that the quality of the software is enhanced and maintained. Brief explanations of software quality, software measurement and software maintenance have been given along with introductory explanations of impact analysis and ripple effect. Future work will include taking the ideas presented here further by drawing up a much more detailed framework of ripple effect and impact analysis implementation throughout the software lifecycle. For this work to be useful guidelines for the practical implementation of the ideas presented need to be drawn up and utilised.

5.0 References


