
Empirical Software Engineering, 4, 241–262 (1999)

Software Evolution and the Code Fault Introduction Process

SEBASTIAN G. ELBAUM elbaum@cse.un1.edu
Department of Computer Science and Engineering, University of Nebraska, Lincoln, Lincoln, NE 68588-0115

JOHN C. MUNSON jmunson@cs.uidaho.edu
Computer Science Department, University of Idaho, Moscow, ID 83844-1010

Abstract. In any manufacturing environment, the fault introduction rate might be considered one of the most meaningful criterion to evaluate the *goodness* of the development process. In many investigations, the estimates of such a rate are often oversimplified or misunderstood generating unrealistic expectations on the prediction power of regression models with a fault criterion. The computation of fault introduction rates in software development requires accurate and consistent measurement, which translates into demanding parallel efforts for the development organization. This paper presents the techniques and mechanisms that can be implemented in a software development organization to provide a consistent method of anticipating fault content and structural evolution across multiple projects over time. The initial estimates of fault introduction rates can serve as a baseline against which future projects can be compared to determine whether progress is being made in reducing the fault introduction rate, and to identify those development techniques that seem to provide the greatest reduction.

Keywords: Software evolution, software measurement, fault introduction process

Introduction

As a software system progresses through a number of sequential builds, faults will be identified during code inspections and software test processes. The code will be changed in an attempt to eliminate the identified faults. Unfortunately the process of code modification is, in itself, a fault prone process. The introduction of new code will offer the opportunity of introducing faults just as was the case with the initial code generation. New faults will be introduced into the code during this evolutionary process. In Figure 1 we can see that for a given build, there are three contributing factors to the fault content of programs.

- There is the initial fault burden at the first build.
- Faults are removed due to inspections and test.
- New faults are introduced as a result of these changes.

Code does not always change just in response to the fault repair process. Some changes to code represent enhancements or changes in the code in response to evolving requirements. These incremental changes will also result in the introduction of still more faults.

The general notion of software test is that the rate of fault removal will generally exceed the rate of fault introduction. In most cases, this is probably true. Some changes are rather more heroic than others. During these more substantive change cycles, it is quite possible that the actual number of faults in the system will rise. We would be very mistaken, then, to

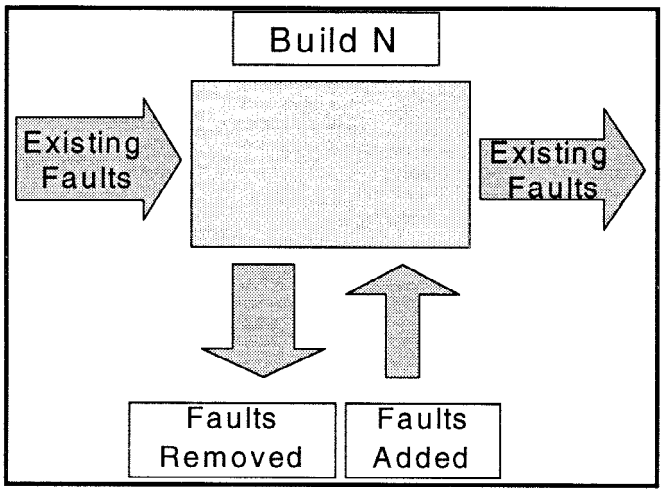

Figure 1. Software evolution and faults.

assume that software test will monotonically reduce the number of faults in a system. This will only be the case when the rate of fault removal exceeds the rate of fault introduction. The rate of fault removal is relatively easy to measure. The rate of fault introduction is much more tenuous. It is directly related to the net change in the code from one sequential software build to the next. Thus, the first step in understanding the nature of the fault introduction process involves the establishment of a methodology for measuring the nature of changes that occur in the build process. In other words, we wish to measure the software evolution process, specifically as this measurement relates to the fault introduction process.

Setting a Measurement Baseline

The measurement of an evolving software system through the shifting sands of time is not an easy task (Munson, 1996; Nikora et al., 1997). Perhaps one of the most difficult issues relates to the establishment of a baseline against which the evolving systems may be compared. We need a fixed point against which all others can be compared. Our measurement baseline also needs to maintain the property that, when another point is chosen, the exact same picture of software evolution emerges, only the perspective changes (Luqi, 1990). The individual points involved in measuring software evolution are individual builds of the system.

This notion is analogous to a topographical mapping problem. Figure 2 shows a hypothetical property located on the side of a mountain. If we wish to obtain a topological survey of this property we must simply pick an arbitrary point on the property against which all other points will be measured. In this example, either point A or point B will do. Regardless of which of the two points we were to choose as a baseline, we would get essentially the same picture of the property.

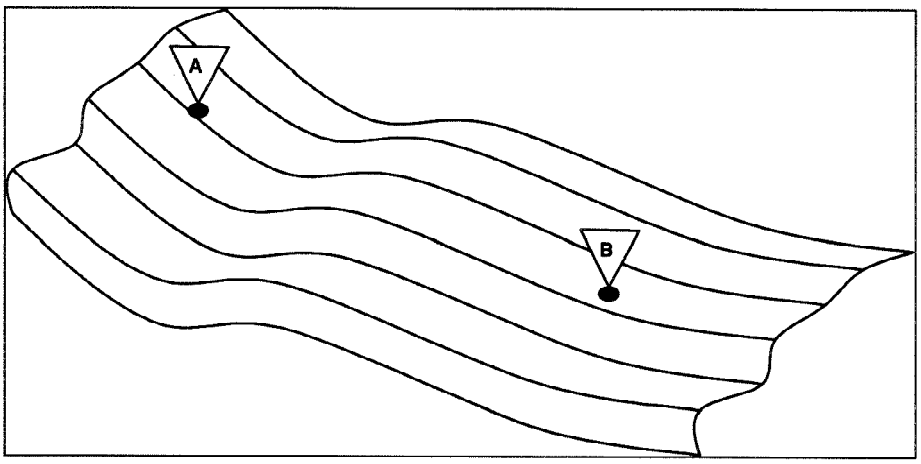

Figure 2. Topology example.

To continue this analogy further, the choice of point A over point B as a baseline is not really arbitrary. If, for example, we wish to build a house towards the upper left on this property, we would certainly choose to use point A as a baseline. This is so that we know that the further away a particular point is on this map, the less accurate will be our estimate as to its elevation relative to the baseline. Very simply, if we had a simple line level and a 10 meter measuring tape, every point more than ten meters away from the baseline would have inaccuracies introduced by the fact that the measuring tape had to be completely moved. The further we are away from the baseline, the greater will be this inaccuracy in measurement.

In the above topological example, we have two attributes that we must measure, height and distance. Further, these two attributes are measured on the same scale, feet or meters. When we measure software systems, we will measure multiple attributes such as statement count, *Stmnts*, and number of cycles in the control flowgraph, *Cycles*. In this case the units of measure are not the same. Ten *Stmnts* and ten *Cycles* are very different things. Essentially all of the software attributes that we wish to measure are assessed by metrics all defined on different scales (Khoshgoftaar and Munson, 1990). Comparing different modules within a software system by using these measurement data is complicated by this fact (Munson, 1995).

In order to measure successive builds of a system, a referent system, or baseline, must be established based on standardized metric values (Munson and Khoshgoftaar, 1990). Standardizing metrics for one particular build is simple. For each metric obtained for each module, subtract from that metric its mean and divide by its standard deviation. This puts all of the metrics on the same relative scale, with a mean of zero and a standard deviation of one. This works fine for comparing modules within one particular build. But when we standardize subsequent builds using the means and standard deviations for those builds a

problem arises. The standardization masks the change that has occurred between builds. In order to place all the metrics on the same relative scale and to keep from losing the effect of changes between builds, all build data is standardized using the means and standard deviations for the metrics obtained from the baseline system. This preserves trends in the data and lets measurements from different builds to be compared.

For each raw metric in the baseline build, we may compute a mean and a standard deviation. Let us denote the vector of mean values for the baseline build as $\bar{\mathbf{x}}^B$ and the vector of standard deviations as $\mathbf{s}^B$. The standardized baseline metric values for any module j in an arbitrary build i, then, may be derived from raw metric values $w^{B,i}$ as

$$z_j^{B,i} = \frac{w_j^{B,i} - \bar{x}_j^B}{s_j^B}$$

We will use the fault index metric (Munson and Khoshgoftaar, 1990b) in the establishment of a suitable baseline. We have developed the methodology for the fault index metric over a number of years. This index is obtained in a two step process from a suite of metrics defined on a set of program structural attributes. First, factor scores on each program module in the baseline, or referent program build, for each set of metrics are created by principal components analysis together with an orthogonal transformation matrix. Next, it would be useful if each of the program modules in a software system could be characterized by a single value representing some cumulative measure of the system structure. To this end, the orthogonal factor scores (domain metrics) are then combined into a single, fault index. Previous research has established that the fault index has properties that will be useful in this regard (Munson and Khoshgoftaar, 1990b; Munson and Khoshgoftaar, 1990; Khoshgoftaar and Munson, 1992; Munson, 1996; Elbaum and Munson, 1998).

The fault index, ρ, of the factored program modules may be represented as

$$\rho_i = \sum_j \lambda_i d_{ij}$$

where λ_i is the eigenvalue associated with the j^{th} factor and d_{ij} is the j^{th} domain metric of the i^{th} program module. The domain metrics are derived from the relationship

$$\mathbf{d} = \mathbf{TZ}$$

where $\mathbf{Z}$ is the matrix of metric z scores for a program and $\mathbf{T}$ is the transformation matrix obtained from the principal components analysis of the z scores. Each of the eigenvalues represents the relative contribution of its associated domain to the total variance explained by all of the domains. In essence, then, the fault index is a weighted sum of the individual domain metrics. In this context, the fault index represents each raw metric in proportion to the amount of unique variation contributed by that metric. As we will see, the real utility of this measure from a software evolution perspective is that it will permit us to measure the amount of change in a system and also the rate of change in measured attributes of a software system.

The fault index has been established as a successful surrogate measure of software faults. It seems only reasonable that we should use it as the measure against which we compare

different builds. Since the fault index is a composite measure based on the raw measurements, it can incorporate represented by *Stmnts, Cycles* and any other raw metrics found to be related to software faults.

By definition, the average fault index, $\bar{\rho}^b$, of the baseline *system* will be

$$\bar{\rho}^b = \frac{1}{N^b} \sum^{N^b} i = 1\rho_i^b = 50,$$

where N^b is the cardinality of the set of modules on build b, the baseline build. The fault index for the baseline build is calculated from standardized values using the mean and standard deviation from the baseline metrics. The fault indices are then scaled to have a mean of 50 and a standard deviation of 10. For that reason, the average fault index for the baseline system will always be a fixed point. Subsequent builds are standardized using the means and standard deviations of the metrics gathered from the baseline system to allow comparisons. The formula for calculating average fault index values for subsequent builds is given as

$$\bar{\rho}^k = \frac{1}{N^k} \sum_{i=1}^{N^k} \rho_i^k,$$

where N^k is the cardinality of the set of program modules in the k^{th} build and ρ_i^k is the fault index for the i^{th} module of that set.

Software Evolution

A software system consists of one or more software modules. As the system grows and modifications are made, the code recompiled and a new version, or build, is created. Each build is constructed from a set of software modules. The new version may contain some of the same modules as the previous version, some entirely new modules and it may even omit some modules that were present in an earlier version. Of the modules that are common to both the old and new version, some may have undergone modification since the last build. When evaluating the change that occurs to the system between any two builds (software evolution), we are interested in three sets of modules. The first set, M_c, is the set of modules present in both builds of the system. These modules may have changed since the earlier version but were not removed. The second set, M_a, is the set of modules that were in the early build and were removed prior to the later build. The final set, M_b, is the set of modules that have been added to the system since the earlier build.

The fault index of the system R^i at build i, the early build, is given by

$$R^i = \sum_{c \in M_c} \rho_c^i + \sum_{a \in M_a} \rho_a^i.$$

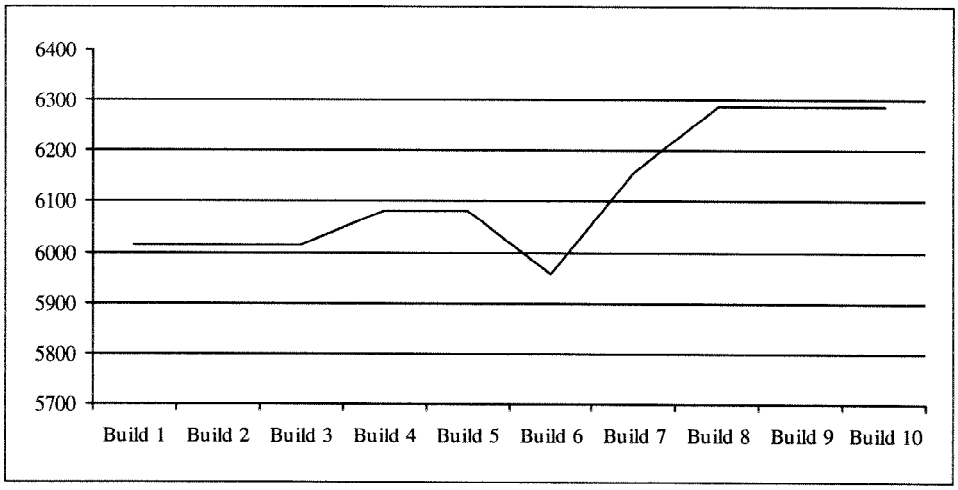

Figure 3. Evolution of fault index over ten successive builds.

Similarly, the fault index of the system R^j at build j, the later build is given by

$$R^j = \sum_{c \in M_c} \rho_c^j + \sum_{b \in M_b} \rho_b^j.$$

The later system build is said to be more fault prone if $R_j > R_i$.

As a system evolves through a series of builds, its fault burden will change. This fault burden is measured by a set of software metrics. One simple assessment of the size of a software system is the number of lines of code per module. However, using only one metric neglects information about the other measurable attributes of the system, such as control flow and temporal complexity. By comparing successive builds on their domain metrics it is possible to see how these builds either increase or decrease based on particular attribute domains. Using the fault index, the fault burden of the overall system can be monitored as the system evolves.

Regardless of which metric is chosen, the goal is the same. We wish to assess how the system has changed, over time, with respect to that particular measurement. The concept of a code delta provides this information. A code delta is, as the name implies, the difference between two builds as to the fault index.

For purposes of demonstration, an embedded real-time system, JTQ, has been evaluated. This is a real time control system of approximately 3500 modules and over 200 KLOC modules (functions) programmed in C. The overall trend in the fault index between ten recent, successive builds is shown in Figure 3.

The pattern shown is quite typical of an evolving software system. In this case, the build labeled as Build 1 is not the initial build. It is the first of a sequence of 10 builds in a very mature embedded software system. In looking at this figure we can see that there are periods of relative quiescence and also periods of great change in the system. The overall

trend is always towards increased intricacy. This particular build sequence was chosen for presentation in that it represents a typical software process problem. At about Build 4 the developers begin to believe that the system has grown to cumbersome. It will need substantial modifications. Build 6 shows the results of this *simplification* effort. There is a relatively dramatic shift downwards in the net complexity of the system (as measured by FI). Builds 7 and 8 represent a rebound from the *simplification* of the code. It was discovered that the *simplification* deleted considerable necessary functionality from the code. This was added back on Builds 7 and 8. On Builds 9 and 10 we see stability returning to the code base. They finally got the system working again. It is interesting to note that the simplified system is now considerably more complex than the system before the *simplification* took place. This is a pattern that we have observed many times in many different software development organizations. In general, though, this misdirection in software development goes unnoticed in that there is no code measurement system in place to evaluate the effect of the *simplification* effort. Most developers, in fact, will swear that the *simplification* was a smashing success. In the absence of a measurement program, we are forced to accept their assessment.

The change in the fault burden in a single module between two builds may be measured in one of two distinct ways. first, we may simply compute the simple difference in the module fault index between build i and build j. We will call this value the code delta for the module, or $\delta_a^{i,j} = \rho_a^j - \rho_a^i$. The absolute value of the code delta constitutes an evolutionary fault index. It will shown shortly that what is important is the absolute measure of the nature that code has been modified. From the standpoint of fault introduction, removing a lot of code is probably as catastrophic as adding a bunch. The new **evolutionary fault index (EFI)**, χ, for module a is simply

$$\chi_a^{i,j} = |\delta_a^{i,j}| = |\rho_a^j - \rho_a^i|.$$

The total net change of the system is the sum of the EFI for a system between two builds i and j is given by

$$\nabla^{i,j} = \sum_{c \in M_c} \chi_c^{i,j} + \sum_{a \in M_a} \rho_a^j + \sum_{b \in M_b} \rho_b^j.$$

The net code delta values and the net EFI for the JTQ system discussed earlier are shown in Figure 4. In this case, the EFI and code delta values are computed between sequential builds.

Figure 4 gives us a much greater perspective on the nature of the *simplification* effort described in Figure 3. The net code deltas for the individual reflect the signed change in FI from build to build. That is, if many modules are removed from the system the net complexity of the system will decrease. That is exactly what happens on Build 4. On the next two builds we can see that a considerable amount of code is coming back into the system. The net code delta is on the rise. Finally the magnitude of the change begins to fall and stabilizes on Builds 9 and 10.

We get a very different view of the system by looking at the EFI. The EFI measures total code activity. It is a measure of the degree of the change. From Figure 4 we can see that the overall change activity gradually begins to increase at Build 2. It reaches its peak frenzy

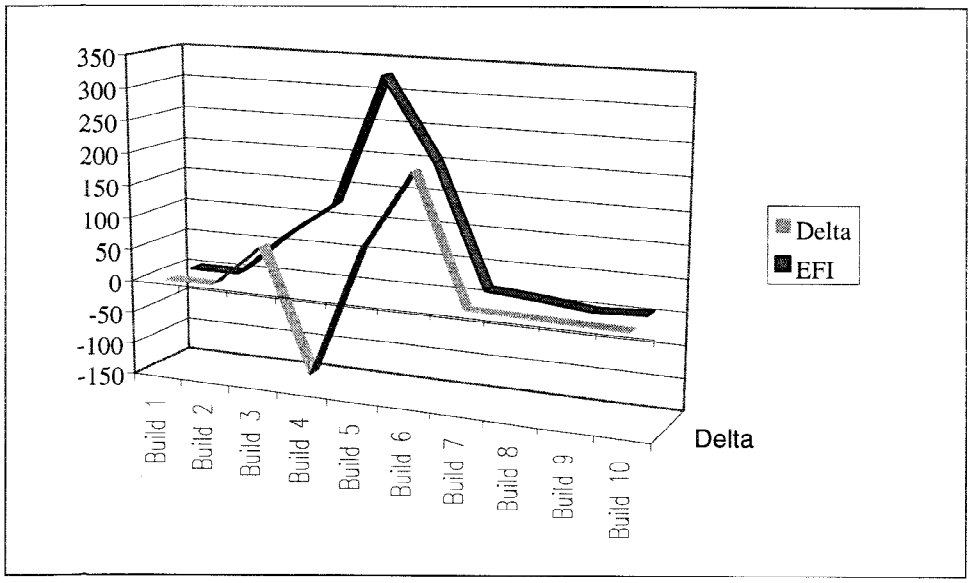

Figure 4. EFI and Code Delta over 10 successive builds.

at Build 6 and then ultimately begins to stabilize at Build 7. In that faults are directly associated with the net change activity and not the degree of change (Elbaum and Munson, 1998), the EFI is a very good index of the rate at which new faults are being introduced into the system. In which case, we can see that Builds 3–7 are those that will require the greatest amount of regression test effort to identify and remove the faults that have been introduced during these relative radical change periods.

With a suitable baseline in place, and the module sets defined above, it is now possible to measure software evolution across a full spectrum of software metrics. We can do this first by comparing average metric values for the different builds. Secondly, we can measure the increase or decrease in the system structure as measured by a selected metric, code delta, or we can measure the total amount of change the system has undergone between builds with the EFI.

A limitation of measuring code deltas is that it doesn't give an indicator as to how much change the system has undergone. If, between builds, several software modules are removed and are replaced by modules of roughly equivalent measurable attributes, the code delta for the system will be close to zero. The overall structure of the system, based on the metric used to compute deltas, will not have changed much. However, the reliability of the system could have been severely effected by the process of replacing old modules with new ones. What we need is a measure to accompany code delta that indicates how much change has occurred. The EFI is a measurement, calculated in a similar manner to code delta that provides this information.

When several modules are replaced between builds by modules of roughly the same structural attributes, code delta will be approximately zero but the EFI will be equal to the sum of the value of ρ for all of the modules, both inserted and deleted. Both the code delta and EFI for a particular metric are needed to assess the evolution of a system.

Software Evolution and the Fault Introduction Process

The principle behind the concept of the fault index is that it serves as a fault surrogate. That is, it will vary in precisely the same manner as do software faults. The fault potential r_i^0 of a particular module i is directly proportional its value of the fault index. Thus,

$$r_i^0 = \rho_i^0 / R^0.$$

where R^0 is simply the sum of all fault index values of each module of the initial system,

$$R^0 = \sum_{i=1}^{N} \rho_i^0.$$

To derive a preliminary estimate for the actual number of faults per module we may make judicious use of historical data. From previous software development projects it is possible to develop a proportionality constant, say k, that will allow the total system fault index to map to a specific system fault count as follows: $F^0 = kR^0$ or $R^0 = F^0/k$. Substituting for R in the previous equation, we find that

$$r_i^0 = k\rho_i^0 / F^0.$$

Initially, our best estimate for the number of faults in module i in the initial configuration of the system is

$$g_i^0 = r_i^0 F^0.$$

As the i^{th} module was tested during the test activity of the first build, the number of faults found and fixed in this process was denoted by f_i^1. However, in the process of fixing this fault, the source code will change. In all likelihood, so, too, will the fault index of this module. Over a sequence of builds, the structure of this module may change substantially. The cumulative EFI in the total system over these j builds will be,

$$\nabla^{0,j} = \sum_{i=1}^{N_j} \nabla_i^{0,j},$$

where N_j is the cardinality of the set of all modules that were in existence over these j builds. The structure of the i^{th} module will have changed over this sequence of builds. Some changes may increase the fault index of this module and others may decrease it. The cumulative EFI of the i^{th} module will be $\rho_i + \nabla_i^{0,j}$.

On the initial build of the system the initial burden of faults in a module was proportional to the fault index of the module. As the build cycle continues the rate of fault introduction

is most closely associate with the EFI. Thus, the proportion of faults in the i^{th} module will have changed over the sequence of j builds, related to its initial fault index value and its subsequent EFI. Its new value will be

$$r_i^j = (\rho_i^0 + \nabla_i^{0,j})/(R^0 + \nabla^{0,j})$$

We now observe that our estimate of the number of faults in the system has changed. On the j^{th} build there will no longer be F^0 faults in the system. New faults will have been introduced as the code has evolved. In all likelihood, the initial software development process and subsequent evolution processes will be materially different. The focus is being changed from a pure development mode to a maintenance mode. This means that there will be a different proportionality constant, say k', representing the rate of fault introduction for the evolving system. For the total system, then, there will have been $F^j = kR^0 + k'\nabla^{0,j}$ faults introduced into the system from the initial build through the j^{th} build. Each module will have had $h_i^j = r_i^j F^j$ faults introduced in it either from the initial build or on subsequent builds. Thus, our revised estimated of the number of faults remaining in module i on build j will be

$$g_i^j = h_i^j - f_i^j.$$

The rate of fault introduction is directly related to the change activity that a module will receive from one build to the next. At the system level, we can see that the expected number of introduced faults from build j to build $j+1$ will be

$$\begin{aligned} F^{j+1} - F^j &= kR^0 + k'\nabla^{0,j+1} - kR^0 + k'\nabla^{0,j} \\ &= k'(\nabla^{0,j+1} - \nabla^{0,j}) \\ &= k'\nabla^{j,j+1} \end{aligned}$$

At the module level, the rate of fault introduction will, again, be proportional to the level of change activity. Hence, the expected number of introduced faults between build j to build $j+1$ on module i will be simply $h_i^{j+1} - h_i^j$.

The two proportionality constants k and k' are the ultimate criterion measures of software development process and software maintenance processes. Each process has an associated fault introduction proportionality constant. If we institute a new software development process and observe a significant change downward in the constant k, then the change would have been a good one. Very frequently, however, software processes are changed because development fads change and not because a criterion measure has indicated that a new process is superior to a previous one. We will consider that an advance in software development process has occurred if either k or k' has diminished significantly for that new process. This will clearly indicate that the rate at which new faults are being introduced is declining either for the initial program development phase or from the evolutionary phase.

Identifying and Counting Faults

As it was shown in the previous section, generating an effective fault index is not enough to compute the fault introduction rate. We also need to count faults consistently in order to compute the fault introduction rate.

Unfortunately, there is no particular definition of just precisely what a software fault is. In the face of this difficulty it is rather hard to develop meaningful associative models between faults and metrics. In calibrating our model, we would like to know how to count faults in an accurate and repeatable manner. In measuring the evolution of the system to talk about rates of fault introduction and removal, we measure in units to the way that the system changes over time. Changes to the system are visible at the module level, and we attempt to measure at that level of granularity. Since the measurements of system structure are collected at the module level (by module we mean procedures and functions), we would like information about faults at the same granularity. We would also like to know if there are quantities that are related to fault counts that can be used to make our calibration task easier.

Simply put, a fault is a structural defect in a software system that may lead to the system's eventually failing (IEEE, 1989; Musa, 1989). In other words, it is a physical characteristic of the system of which the type and extent may be measured using the same ideas used to measure the properties of more traditional physical systems. Structural defects in jet turbine blades, for example, are well defined and their consequences well understood. There are precise standards for recognizing these faults and assessing their potential impact on the successful operation of the jet engine of which they are a part. Unfortunately, we have no such industry standard for recognizing software defects nor an understanding of the consequences of these faults.

The Fault Counting Process

The first step in the measurement of software faults for this study was to set up the necessary documentation flow in order to collect all the information that was required. Change report forms were used by a developer to propose a change that was needed in response to failures or/and enhancements or other kind of modifications. A committee reviewed these forms and if approved, the developer could modify the code. Once the modifications were completed, the form could be check-in by a developer.

At that point, a new script was coupled to the check-in process to search for the code modified by the developer in response to that change form. The search is done mainly within the structure of the existing configuration control system, RCS. The raw metrics from the code are extracted and the fault index is computed using the corresponding baseline. This new data is included in a third new section of the change form. This process occurs automatically to ensure consistency and it proved to be more efficient. At the same time, a mail message notifying the completion of this process is sent to the group in charge of the fault tracing and counting.

The existing mechanisms for fault tracking contained descriptions of the failures at varying levels of detail, as well as descriptions of what was done to correct the fault(s) that caused the failure. Detailed information regarding the underlying faults (e.g., where were the code changes made in each affected module) was generally unavailable from the problem reporting system. In order to count faults, we needed to develop a method of identification that is repeatable, consistent, and identifies faults at the same level of granularity as our structural measurements. This type of fault is simple to count. It occurs only in one module.

In identifying and counting faults, we must deal with faults that span only one module as well as those that span several.

The fault counting process that we instituted starts when a mail message is received from the code measurement process. That message includes the change form and the necessary links to the version of the file that was measured. The first step is to determine whether the code change was in response to a fix or if it was in response to some other kind of event. To determine whether it is a fault fix, there is a need to analyze the change form exhaustively and complement it with the assistance of the developers involved in it.

Once the change is recognized as a response to a fault, the fault tracing begins. The fault needs to be traced to its origin, where it was originally introduced. In order to trace the faults, previous versions of the faulty module have to be carefully analyzed one by one, until the fault is found. It is obvious that this process is very demanding but, as it will be shown, it pays off in terms of consistency and repetitiveness. Some tools such as differential comparator (e.g., Unix "diff") are used to make this job easier. The sections of code highlighted by the comparator will indicate where the faults might be located. Some of these differences will reflect enhancements or other type of activity not due to faults, but this discrimination can be done with the information provided in the change forms and with the developers assistance.

After completing the last step, we still had to identify and count the faults—the results of the differential comparison cannot simply be counted up to give a total number of faults. In order to do this, we used a taxonomy for identifying and counting faults that is based on the types of changes made to the software to repair the faults associated with failure reports (Nikora and Munson, 1998). An example of why this is so is found within the following discussion of the taxonomy.

Note that this taxonomy differs from many others in that it does not seek to identify the root cause of the fault. Although identifying the root causes of faults is important in improving the development process (Chillarege et al., 1992; IEEE, 1993), it is first necessary to identify the faults. This is the specific issue addressed by this taxonomy. We have found that this taxonomy has allowed us to successfully identify faults in the software used in the study in a consistent manner at the appropriate level of granularity. Briefly, there are three categories of faults—faults associated with variables, faults associated with constants, and control flow faults.

1. Faults Associated with Variables

    1.1) Definition and use of new variables in a module

    1.2) Re-definition of existing variables

    1.3) Deletion of an existing variable

    1.4) Change of value in an existing variable assignment statement

2. Faults Associated with Constants

    2.1) Definition and use of new constants in a module

    2.2) Re-definition of existing constants

2.3) Deletion of an existing constant

3. Control Flow Faults

3.1) Addition of new source code block

3.2) Deletion of erroneous conditionally-executed path(s) within a code block

3.3) Addition of execution path(s) within a source code block

3.4) Re-definition of execution condition

3.5) Removal of source code block

3.6) Incorrect order of execution

3.7) Addition of a procedure or function

3.8) Removal of a procedure of function

3.9) Call to an external function and passing incorrect argument

3.10) Erroneous return value

3.11) Erroneous Boolean operator

3.12) Erroneous Pointer

3.13) Allocated memory not initialized

3.14) Allocated memory not freed

Faults can be more complex than the situations indicated above—it is possible to have faults on top of faults. We can decompose these more complicated situations into simpler ones that can be handled by application of the rules given above. For instance, the order of execution of two blocks may be changed, and one of these blocks may also be changed to include a reference to a new variable. In addition, special care was given to faults that spanned across modules. For instance, a wrong definition in a header file that was included and used in two different modules was counted as two separate faults. Each module fault count was incremented by one because after compilation, the program had two faults that were likely to be executed. In general, all faults that propagated through different modules received a similar treatment. Our principal concern in the proper enumeration of faults is that each fault be represented as it occurs in the compiled code. If a fault is introduced into a header file and this header file is subsequently included in five modules, then each of the compiled modules will have this fault at run time.

Estimating the Fault Introduction Rate

We would now like to turn our attention to the task of estimating the rate of fault introduction, k', for the software maintenance activity. The automatic source code measurement process turned out to be an almost trivial task once the tools were all in place. It was highly mechanized and transparent to all developers.

Table 1. Metric definitions.

Metric	Description
Comm	Total comment count
ExStmt	Executable statements
NonEx	Non executable statements such as data declaration
N_1	Total number of operands
η_1	Unique operands
N_2	Total number of operators
η_2	Unique operators
η_3	Unique operators with overloading
Nodes	Number of nodes in the module control flowgraph
Edges	Number of edges in the module control flowgraph
Paths	Number of distinct paths in the module control flowgraph
MaxPath	Maximum path length in the module control flowgraph
$\overline{Path}$	Average path length in the module control flowgraph
Cycles	Total cycle count in the module control flowgraph

It allows us to gather all the information automatically and independently of the developers. The first of those tools is called CMA (C Metric Analyzer). CMA generates a working set of complexity metrics that are known to be highly correlated to software faults. The driving forces in the development of this measurement tool have been to construct a suitable surrogate measure for software faults and to be able to plug-in the development process as a part of the configuration control system already in place. Each of the metrics included in the tool explains a significant amount of variation in the concomitant criterion measure of software faults. The validation metrics (Elbaum and Munson, 1998) produced by the CMA tool are shown in Table 1.

The second tool is PCA/FI (Principal Components Analysis/Fault Index). This tool performs the principal component analysis on a baseline build and stores the information that is needed to compute the fault index metric for successive builds. When the tool is invoked for a baseline build, it will produce the vector $\bar{\mathbf{x}}^B$ of metric means, $\mathbf{s}^B$ their standard deviation, the transformation matrix $\mathbf{T}$, and the domain metric values for the baseline build. When the tool is invoked in subsequent non-baseline builds, it will transform the new metric value using the $\bar{\mathbf{x}}^B$, $\mathbf{s}^B$ and $\mathbf{T}$ of the baseline build.

Four different sources of variation were identified in the 14 metrics collected on the JTQ system by our principal components analysis tool, PCA/FI. Each of these sources of variation was mapped to orthogonal components by the principal component analysis. Component

Table 2. The fault index and the EFI.

Module number	FI Build 1	EFI Builds 1–2	EFI Builds 2–3	EFI Builds 3–4
1	50.75	0	0	0
2	61.25	1.99	0	1.18
3	53.12	0.57	0	0
4	51.10	0	0	0
5	49.08	0	–	–
6	49.92	0	0	0
7	55.08	0	0	0
8	46.32	1.75	0.76	0
9	56.82	0.61	0	1.89
10	–	–	44.47	0

1 had associated with it those metrics related to volume or size and represented 65% of the observed variation. Component 2 consisted of the metrics associated to the flow control dimension and represented 10% of the observed variation. Component 3 consists from the both operator metrics. Finally, Component 4 represents the non-executable statements in the module such as data structure declarations. Each of these components represents an incremental source of variation. Our objective in measurement is to identify as many distinct sources of measurement that explain sources of variation in the criterion measure of software faults. Some faults are due to the size of the program. Some faults result from the control complexity of the program. Some faults result from the data structures complexity of the program, and so forth.

The third tool is the software evolution tool evolution tool, EVOLV. This tool allows the automatic comparison of the complexities of two builds. EVOLV uses the fault index values stored by PCA/FI for each one of the software modules of each build. It seeks to match equivalent modules for both builds. It takes into consideration what modules were added, removed or modified and it computes the code delta and EFI for the latest build.

In Table 2, the fault indexes computed by PCA/FI over the baseline build (build 1) are presented for ten sample or indicative program modules of the JTQ system. The third, fourth and fifth column of Table 2, presents the evolutionary fault index computed between consecutive builds. Most of the modules are presented across all builds, presenting slight change in their EFI due to minor fixes. Module ten was added in a later build and contains almost all of the EFI between builds 2 and 3. Intuitively, module 10 constitutes new code that might have high fault burden. On the other hand, module 5 was removed from the system in build 3 which it will increase the total or net EFI.

These tools were incorporated into the development process. As it can be seen in Figure 5, the tools are part of a flow of metric information that occurs transparently and automatically.

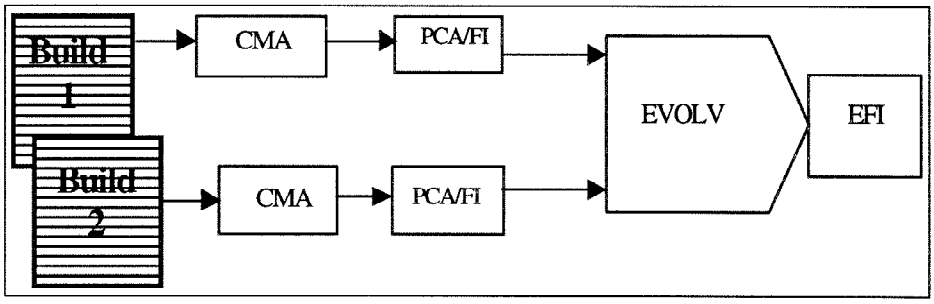

Figure 5. Software tools and processes.

After all the software change requests intended to be in a build are completed, the build process starts. At that exact time, a script launches the measurement tool over the modules that have been modified and stores the data. When the measurement is complete, PCA/FI computes fault index for all the modules that have changed. At last, the script locates the previous build and calls the evolution tool. The evolution tool searches in the measurement files for the fault index value for every module involved in the builds and then computes the code delta and the EFI.

An Empirical Determination of k'

We will now describe the application of these tools and methodologies to obtain an empirical assessment for the rate of fault introduction, k', in an actual commercial software development organization. To this end, we installed all of our tools to work in conjunction with the RCS configuration control system. The CMA metric tool measured each module as it was checked back into the RCS system having been modified by a developer. The PCA/FI tool then computed a baselined fault index value for the new metric values. Before each build, the EVOLV tool was invoked by the build process to compute the net change across all modules between the current build and the previous one.

In order to obtain precise measures for our experimental criterion variable, faults, each fault had to be extracted from a mass of failure reports. This fault tracing and counting process was not an easy job. On the contrary, it required a major time investment in order to do it correctly. Over 650 failure reports were studied during a period of 8 months. It was not possible to trace all 650 failure reports back to their point of origin. We extracted a 10% random sample of these change report for further investigations. From this sample, we eliminated those failure reports that reported on the same fault condition.

Each of the fault reports was traced back through the maze of the revision control system until the version where the fault was introduced was found. This process required in most cases, to analyze the differences between several versions of the module before reaching

Table 3. Fault counting and analysis.

Code			**Fault**
Original	**Modification**	**Fix**	
. . .	. . .	. . .	Type:
cool_fci.pin[0] = 0;	cool_fci.pin[0] = 0;	cool_fci.pin[0] = 0;	Control Flow Fault
cool_fci.pin[1] = 0;	cool_fci.pin[1] = 0;	cool_fci.pin[1] = 0;	Subtype:
cool_fci.pin[2] = 0;	cool_fci.pin[2] = 0;	cool_fci.pin[2] = 0;	3.3. Addition of execution path(s) within a source code block.
cool_fci.pin[3] = 0;	cool_fci.pin[3] = 0;	cool_fci.pin[3] = 0;	
cool_fci.pin[4] = 0;	cool_fci.pin[4] = 0;	cool_fci.pin[4] = 0;	
cool_fci.pin[5] = 0;	cool_fci.pin[5] = 0;	cool_fci.pin[5] = 0;	Counting Rule:
. . .	**cool_fci.pin[6] = 1;**	cool_fci.pin[6] = 1;	One fault is counted for each execution path within a conditional-execution block that is added in response to a fault report. Again, this assumes that the execution conditions for the original execution paths do not change.
	cool_fci.pin[7] = 0;	**if (turn_cool == 1)**	
	. . .	cool_fci.pin[7] = 0;	
		. . .	
Introduction of fault		**Fault Removal and Counting: 1 Fault**	

the one in which the fault introduction occurred. Each fault took an average of 9 hours to be traced and counted. On several occasions, the fault could not be traced to its origin in that the older versions of the file were not available in the revision control system. A total of 41 faults were successfully traced to their points of origin for builds of the system whose source code deltas remained in the RCS system. Then, the EFI was computed between the version where the fault(s) was introduced and its previous version.

An example of this procedure is presented in Table 3. The original version of the module is on the first column. The code where the fault was introduced is on the second column. The EFI was computed between those two versions of the module. In the third column, the code after the fix is presented. The type of fault is specified in the fourth column. Observe that the fault counting and classification is performed based on the code that needs to be introduced in order to fix the fault. The fixed code is the target of the fault counting. By performing the counting in this manner, faults introduced by the added code or by the lack of code can be accounted properly.

In each of these modules, there may have been more than one fault. Table 4 presents the list of the modules where the faults were found. A total of 41 faults were counted in 23 modules. The EFI measure for each of the modules is presented in the third column of Table 4.

We then regressed the EFI measures against the fault measures for each of these modules. The regression model was developed without a constant term. We assume that there is no intrinsic constant increment for the simple effect of change. Faults are distinctly related to

Table 4. Faulty modules.

	Module ID	EFI	Fault
1	xxx_notify	0.030	1
2	xxx_log_xxx_change	3.028	2
3	xxx_diag_init	6.107	4
4	xxx_write_xxx	1.376	1
5	xxx_send_change	0.964	1
6	xxx_parse_xxx_red	1.795	1
7	xxx_ckeck_xxx_diags	1.410	2
8	xxx_run_xxx_diags	0.502	2
9	xxx_dummy_xxx_msg	0.642	1
10	xxx_identify_xxx_parity	0.028	1
11	xxy_write_xxy_eeprom	3.581	2
12	xxy_file_xxy_1seek	4.439	3
13	xxy_reconstruct_xxy	5.465	3
14	xxy_resp_convert	6.413	3
15	xxy_block_used	4.256	2
16	xxy_reassign_block	6.751	2
17	xxy_explore_memory	4.142	2
18	xxy_explode_ai	0.341	1
19	xxz_evaluate_respond	0.341	1
20	xxz_send_xxz	5.486	1
21	xxz_init_qry_xxz_response	1.991	1
22	xxz_create_xxz_msg	0.045	1
23	xxz_format_xxz_msg	4.665	3
Total		**63.80**	**41**

change effects only. The regression ANOVA is shown in Table 5 below. From this table we can see that there was a significant ($p < 0.05$) linear relationship between the EFI measure and the identified faults.

The regression model is shown in Table 6. The coefficient, 0.500, of the regression model is our estimate for the rate of fault introduction, k'. From this value, it is clear that with the current maintenance process in place, the rate of fault introduction is relatively high. Even for relative small changes, with the EFI value of 10 or less, a potentially large number

Table 5. Regression analysis of variance.

Source	Sum-of-Squares	DF	Mean-Squares	F-Ratio	P
Regression	73.858	1	73.85	94.79	0.000
Residual	17.142	22	10.77		

Table 6. Regression model.

Effect	Coefficient	Std Error	T	P(2-Tail)
EFI	0.500	0.051	9.73	0.000

Table 7. Regression statistics.

N	Multiple R	Squared multiple R	Standard error
23	0.901	0.812	0.883

of faults are introduced into the system. These faults must be removed, at relatively great expense, during the testing process.

The regression model developed from these data is apparently a very good one. The Multiple R for this model was 0.901 as can be seen in Table 7. This means that we were able to account for more than 80% of the variation of the faults with the EFI. This result is quite consistent with our investigation of the rate of fault introduction on the Cassini spacecraft mission data system at JPL (Nikora and Munson, 1998). As is typical of software engineering measurements of software quality, the costs of collecting fault data are very high. Each of these data points was carefully extracted from a failure database by hand. While the number of observations is small, 41 faults were counted across 23 modules, the cost of measurement was extremely high. We can clearly see from Table 5 and Table 6, however, that our experimental objectives were achieved. The standard error for the EFI regression coefficient is quite small.

It is clear that the regression model will lead to valuable prediction of software faults. That, in itself, is worthwhile. What is more important, however, is that we now have a measure, k', of the software evolution process. Should we elect to modify the software development process at some time in the future, we would expect to be able to show that the new, "improved" software process was superior to our current development methodology in that new regression models would should new values for $k' < 0.5$. We would then know

that we had clearly made positive strides in the direction of process improvement.

Conclusion

There is a distinct and a strong relationship between software faults and measurable software attributes. This is in itself not a new result or observation. The most interesting result of this current endeavor is that we also found a strong association between the fault introduction process over the evolutionary history of a software system and the degree of change that is taking place in each of the program modules. We also found that the direction of the change had an effect on the number of faults inserted. Some changes will have the potential of introducing very few faults while others may have a serious impact on the number of latent faults. Different numbers of faults may be inserted, depending upon whether code is being added to or removed from the system. Further, it is possible to exploit these data to estimate the *rate* of fault introduction. This measure of the rate of fault introduction is a direct function of the software development process methodology. A change in software process methodology should yield a ***decrease*** in the rate of fault introduction if the change is a beneficial one.

One of the most disturbing aspects of this study was the extreme difficulty we had in measuring the faults. There is a general disinclination on the part of all software developers to value the importance of recording software fault data. The data that we were able to accurately identify were obtained ex post facto. They were very expensive to collect. If we are to be able to deal effectively with measuring software process change, then we must collect accurate criterion measures for evaluating this process.

In order for the measurement process to be meaningful, the fault data must be very carefully collected. In this study, the data were extracted ex post facto as a very labor intensive effort. Since fault data cannot be collected with the same degree of automation as much of the data on software metrics being gathered by development organizations, material changes in the software development and software maintenance processes must be made to capture these fault data. Among other things, well-defined fault standard and faulty taxonomy must be developed and maintained as part of the software development process. Further, all designers and coders should be thoroughly trained in its use. A viable standard is one that may be used to classify any fault unambiguously. A viable fault recording process is one in which any one person will classify a fault exactly the same as any other person.

The whole notion of measuring the fault introduction process is its ultimate value as a measure of software process. The software engineering literature is replete with examples of how software process improvement can be achieved through the use of some new software development technique. What is almost absent from the same literature is a controlled study to validate the fact that the new process is meaningful (Fenton, 1994). The techniques developed in this study can be implemented in a development organization to provide a consistent method of measuring fault content and structural evolution across multiple projects over time. The initial estimates of fault introduction rates can serve as a baseline against which future projects can be compared to determine whether progress is being made in reducing the fault introduction rate, and to identify those development techniques that seem to provide the greatest reduction.

Acknowledgments

This work was supported in part by the Storage Technology Corporation of Louisville, Colorado and by the National Science Foundation. The authors acknowledge the helpful comments from three anonymous referees.

References

Chillarege, R., Bhandari, I., Chaar, J., Halliday, M., Moebus, D., Ray, B., and Wong, M. Y. 1992. Orthogonal defect classification—A concept for in-process measurement. *IEEE Transactions on Software Engineering* 18(11): 943–946.

Elbaum, S. G., and Munson, J. C. 1998. A standard for the measurement of C complexity attributes. Technical report: TR-CS-98-02, Software Engineering Testing Lab, University of Idaho.

Elbaum, S. G., and Munson, J. C. 1998. Code churn: A measure for estimating the impact of code change. *Proceedings of the International Conference on Software Maintenance*, 24–31.

Fenton, N. E. 1994. Software measurement: A necessary scientific basis. *IEEE Transactions in Software Engineering* 20(3): 199-206.

IEEE Standard Dictionary of Measures to Produce Reliable Software. 1989. IEEE Std. 982.1. Institute of Electrical and Electronics Engineers.

IEEE Standard Classification for Software Anomalies. 1993. IEEE Std. 1044. Institute of Electrical and Electronics Engineers.

Khoshgoftaar, T. M., and Munson, J. C. 1990. Predicting software development errors using complexity metrics. *Journal on Selected Areas in Communications* 8: 253–261.

Khoshgoftaar, T. M., and Munson, J. C. 1992. A measure of software system complexity and its relationship to faults. *Proceedings of the International Simulation Technology Conference* San Diego, CA, 267–272.

Luqi. 1990. A graph mode for software evolution. *IEEE Transactions on Software Engineering* 16(8): 917–920.

Munson, J. C., and Khoshgoftaar, T. M. 1990. Regression modeling of software quality: An empirical investigation. *Journal of Information and Software Technology* 32: 105–114.

Munson, J. C., and Khoshgoftaar, T. M. 1990. The relative software complexity metric: A validation study. *Proceedings of the Software Engineering Conference*. Cambridge, UK, 89–102.

Munson, J. C., and Khoshgoftaar, T. M. 1992. The detection of fault-prone programs. *IEEE Transactions on Software Engineering* 18(5): 423–433.

Munson, J. C. 1995. Software measurement: problems and practice. *Annals of Software Engineering* J. C. Baltzer AG, (1): 255–285.

Munson, J. C. 1996. Software faults, software failures, and software reliability modeling. *Information and Software Technology* N38: 687–699.

Munson, J. C., and Werries, D. S. 1996. Measuring software evolution. *Proceedings of the 1996 IEEE International Software Metrics Symposium*, Berlin, Germany, 41–51.

Musa, J. 1989. Faults, failures and a metrics revolution. *IEEE Software* 6(2): 85–91.

Nikora, A. P. 1998. Software system defect content prediction from development process and product characteristics. Doctoral dissertation, Department of Computer Science, University of Southern California.

Nikora, A. P., Schneidewind, N. F., Munson, J. C. 1997. V&V issues in achieving high reliability and safety in critical control system software. *Proceedings of the International Society of Science and Applied Technological Conference*, Anaheim, California, 25–30.

Nikora, A. P., and Munson, J. C 1998. Software evolution and the fault process. *Proceedings of the twenty third annual software engineering workshop*, NASA/Goddard Space Flight Center (GSFC) Software Engineering Laboratory (SEL).

Nikora, A. P., Munson, J. C. 1998. Determining fault insertion rates for evolving software systems. *Proceedings of the 1998 IEEE International Symposium of Software Reliability Engineering*, Paderborn, Germany, IEEE Computer Society Press.

Dr. Sebastian Elbaum is an Assistant Professor of Computer Science and a member of the J. D. Edwards Honors Program at the University of Nebraska-Lincoln. He received the Ph.D. degree in Computer Science from the University of Idaho, the MS in Science with a software orientation from the same institution and a B.S. in Systems Engineering from the Universidad Catolica of Cordoba, Argentina. His reserach interests include software development processes, measurement, testing, reliability and security. He is a member of the IEEE, the IEEE Computer Society and the IEEE Reliability Society.

Dr. John C. Munson is a Professor of Computer Science at the University of Idaho. He has been actively engaged in research and publication in the areas of software reliability engineering, software measurement, computer security, and software maintenance. He is a member of the Association for Computing Machinery, the IEEE, the IEEE Computer Society and the IEEE Reliability Society. He has been closely associated with the IEEE International Symposium on Software Reliability and the IEEE International Conference on Software Maintenance serving as a member of the program committee and also as program chair for these conferences. He is currently funded for reliability research by the Jet Propulsion Laboratory and for research in software security by the National Science Foundation.