NISS

Does Code Decay? Assessing the Evidence from Change Management Data

Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus

Technical Report Number 81
March, 1998

National Institute of Statistical Sciences
19 T. W. Alexander Drive
PO Box 14006
Research Triangle Park, NC 27709-4006
www.niss.org

Does Code Decay? Assessing the Evidence from Change Management Data

Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, Audris Mockus

Abstract

A central feature of the evolution of large software systems is that change — which is necessary to add new functionality, accommodate new hardware and repair faults — becomes increasingly difficult over time. In this paper we approach this phenomenon, which we term *code decay*, scientifically and statistically. We define code decay, and propose a number of measurements (code decay indices) on software, and on the organizations that produce it, that serve as symptoms, risk factors and predictors of decay. Using an unusually rich data set (the fifteen-plus year change history of the millions of lines of software for a telephone switching system), we find mixed but on the whole persuasive statistical evidence of code decay, which is corroborated by developers of the code. Suggestive indications that perfective maintenance can retard code decay are also discussed.

S. G. Eick is with Bell Laboratories.
T. L. Graves is with the National Institute of Statistical Sciences and Bell Laboratories.
A. F. Karr is with the National Institute of Statistical Sciences.
J. S. Marron is with the University of North Carolina at Chapel Hill.
A. Mockus is with Bell Laboratories.

I. Introduction

Because the digital bits that define it are immutable, software does not age or "wear out" in the conventional sense. In the absence of change to its environment, software can function essentially forever as it was originally designed. However, change is not absent but ubiquitous, in two principal senses. First, the hardware and software environments surrounding a software product do change: for example, hardware is upgraded, or the operating system is updated. Second, and equally important, the required functionality — both features and performance — changes, sometimes abruptly. For example, a telephone system must, over time, offer new features, become more reliable and respond faster.

Necessarily, then, the software itself must be changed, through an ongoing process of maintenance. As part of our experience with the production of software for a large telecommunications system, we have observed a nearly unanimous feeling among developers of the software that the code degrades through time, and maintenance becomes increasingly difficult and expensive.

Whether this *code decay* is real, how it can be characterized, and the extent to which it matters are the questions we address in this paper. The research reported here is based on an uncommonly rich data set: the entire change management history of a large, fifteen-year old real-time software system for telephone switches. Currently, the system comprises 100,000,000[1] lines of source code (in C/C++ and a proprietary state description language) and 100,000,000 lines of header and make files, organized into some 50 major subsystems and 5,000 modules. (For our purposes, a module is a directory in the source code file system, so that a code module is a collection of several files. This terminology is not standard.) Each release of the system consists of some 20,000,000 lines of code. More than 10,000 software developers have participated.

We begin, in §II, with a brief discussion of the software change process and the change management data with which we work. The handling, exploration and visualization of these data are important issues in their own right, and are treated in [1].

In §III, we propose a conceptual model for code decay: a unit of code (in most cases, a module) is decayed if it is *harder to change than it should be*, measured in terms of effort, interval and quality. Associated with the model is a compelling medical metaphor of *software as patient*, which enables one to reason in terms of causes, symptoms, risk factors and prognoses.

The scientific link between the model and the conclusions is a series of code decay indices (CDIs) presented in §IV, which quantify symptoms or risk factors (and so are like medical tests) or predict key responses (a prognosis). The indices introduced here are directly relevant to the statistical analyses that follow; many others could be formulated and investigated.

Our four principal results treat specific manifestations of decay. Three of these results are evidence that code does decay: (1) *the span of changes*, which is shown to increase over time (§V-A); (2) *breakdown of modularity*, which is exhibited by means of network-style visualizations (§V-B); (3) *fault potential*, the likelihood of changes to induce faults in the software system – in §V-C, we show that the distribution of faults is explained by the distribution of large, recent changes to the software. The fourth quantifies the impact of decay, in the form of (4) *prediction of effort* required to make a change, using code decay indices that encapsulate characteristics of changes (§V-D).

Related Work

Early investigations of aging in large software systems, by Belady and Lehman [2], [3], [4], reported the near-impossibility of adding new code to an aged system without introducing faults. Work such as [5] on software maintenance for Cobol programs running on an IBM online transaction processing system addressed program complexity, modularity and modification frequency as explanatory variables, but found that these variables accounted only for 12% of the variation in the repair maintenance rate.

[1]Numbers are approximate.

Bendifallah and Scacchi in [6] consider software maintenance and its effect on cost, interval and quality. Particularly noteworthy because of its historical summary of large scale software development is [7]. Kemmerer and Ream survey empirical work on software maintenance [8].

Our conceptualization of code decay in medical terms was inspired by Parnas [9]. In work related to our fault CDI, Ohlsson and Alberg [10] identify fault-prone modules in switching system software.

Two early fundamental papers relating software data collection and its analysis are [11], [12].

II. Changes to Software

Our definition of a change to software is driven by the data that are available: a change is *any alteration to the software recorded in the change history data base.* The specific data with which we deal are described in §II-B and §V.

The changes we study fall naturally into three main classes (see [13] and [14]) that define the evolution of a software product. *Adaptive changes* add new functionality to a system (for example, caller ID in a telephone switch), or adapt the software to new hardware or other alterations in its environment. *Corrective changes* fix faults in the software. *Perfective changes* are intended to improve the developers' ability to maintain the software without altering functionality or fixing faults. Perfective maintenance has also been called "maintenance for the sake of maintenance" or "re-engineering."

A. The Change Process

For the system we study, changes to the source code follow a well-defined process. *Features* (for example, call waiting or credit card billing) are the fundamental requirements units by which the system is extended.

Changes that implement a feature or solve a problem are sent to the development organization as *Initial Modification Requests* (IMRs); implementation of a feature typically requires hundreds of IMRs. The supervisor responsible for the IMR distributes the work to the developers. Developers implementing parallel changes (as in [15]) must wait for unavailable files.

Each IMR generates a number of *Modification Requests* (MRs), which contain information representing the work to be done to each module. (Thus, an IMR is a problem, while an associated MR is all or part of the solution to the problem.) To perform the changes, a developer "opens" the MR, makes the required modifications to the code, checks whether the changes are satisfactory (within a limited context, i.e., without a full system build), and then submits the MR. Code inspections and integration and system tests follow.

An editing change to an individual file is embodied in a *delta*: the file is "checked out" of the version management system, edited and then "checked in." Lines added and lines deleted by a delta are tracked separately. (To change a line, a developer first deletes it, then adds the new version of the line.[2])

A major organizational paradigm shift (see [16]) for one of the organizations working on the system during its lifetime has been a transition from developer ownership of modules (with a feature implemented by all developers who own modules that are touched) to developer ownership of features, with the feature owner(s) making changes wherever necessary. Implications of this are discussed in §V-A and §V-D.

B. Change Management Data

Data pertaining to the change history of the code itself reside in a version management system, which tracks changes at the feature, IMR, MR and delta levels. Within the version management system, the structure of the changes is as follows (see Figure 1).

[2]This preserves the capability to build earlier versions of the software.

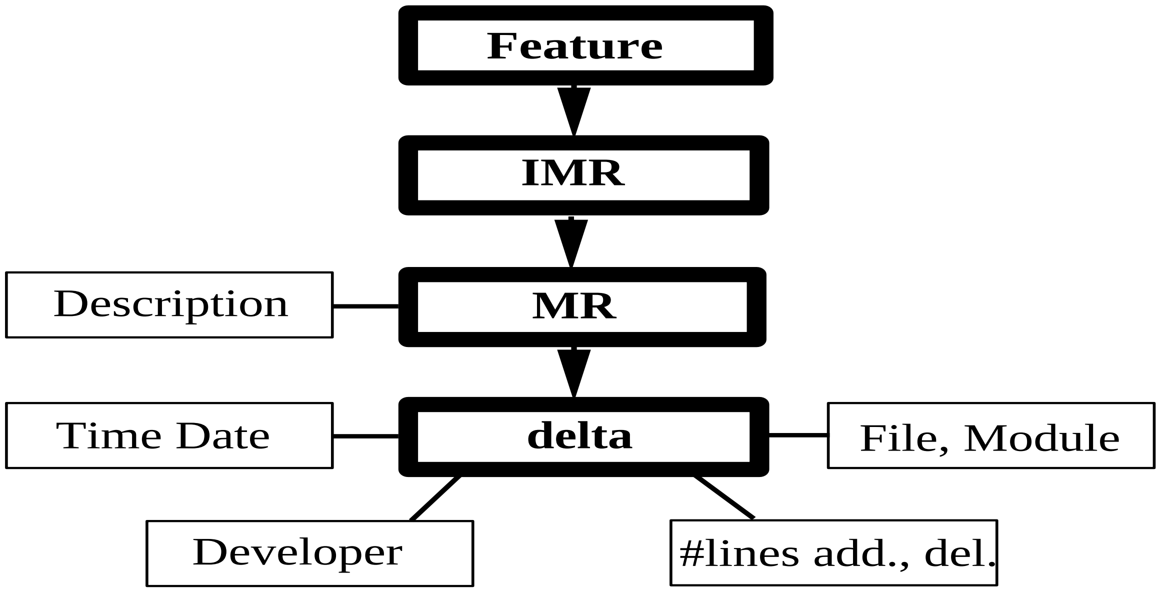

Fig. 1. Changes to the code (bold boxes) and associated data fields.

TABLE I
SUMMARY OF CHANGE DATA

Data Element	Delta	MR	IMR	Feature
What/How Many	D	A	A	A
Who	D	D		
Why		D*	D*	
When	D	D	D	D
How Long		D	D	A
How Much Effort				D

Each IMR has an extensive record containing priority, date opened and closed, point in the development process when it was initiated (requirements, design, coding, testing, field operation), and a number of other fields (89 in all).

Data for each MR include the parent IMR, dates and affected files, and an English text abstract describing the change and the reasons for it. There is no explicit format on how and what information is entered in the abstract; the purpose is for other developers to understand what change was made and why.

The data for each delta list the parent MR and the date and time when the change was submitted to the version management system as well as numbers of lines added, deleted, and unmodified by that change.

Desirable questions for change data to answer are: *WHAT* files were changed, and which lines were added and deleted? *HOW MANY* modules, files, and lines were affected? *WHO* made the change? *WHY* was the change made — did it add new functionality to the code or fix a fault? *WHEN* was the change made? *HOW LONG* did the change take, in calendar time? *HOW MUCH EFFORT* did the change require, in developer-hours?

The extent to which the version management database satisfies these requirements, and at which levels of aggregation of changes, is shown in Table I. In this table, "D" indicates items directly in the database, while "A" denotes items obtained by aggregation over constituent software sub-units. Elements denoted by "D*" have problematic aspects discussed in [1].

III. A Conceptual Model for Code Decay

In this section, we explore a medical metaphor: software suffering from decay can be thought of as diseased. After defining code decay in §III-A, we list some causes of the decay disease in §III-B. The software "patient" may exhibit the symptoms (§III-C), which, as with medical symptoms, suggest that code decay is present. *Risk factors* (§III-D) are reasons for concern about decay, even in the absence of symptoms.

A. What is Code Decay?

Code is decayed if
it is *more difficult to change than it should be*, as reflected by three key responses: (1) *COST* of the change, which is effectively only the personnel cost for the developers who implement it; (2) *INTERVAL* to complete the change — the calendar/clock time required; and (3) *QUALITY* of the changed software.

In the system we study, the interval and quality responses are constrained — schedules must be met and quality standards must be attained, so to a significant extent the key question becomes the cost (effort) necessary to achieve the requisite interval and quality. Even so, interval and quality merit study. Prediction of interval, for example, is crucial in resource allocation decisions. Similarly, quality during the maintenance process is measurable, in terms of errors or unexpected behavior introduced into the system (but later removed).

Several points should be noted. First, code decay is a temporal phenomenon, and it may be useful to add a "more difficult to change *than it used to be*" phrase to the definition.

Second, not all increase in difficulty results from decay: it is possible that the inherent difficulty of the desired changes is increasing.

Third, decay is distinct from the ability of the software to meet requirements: code can be "correct" and still be decayed, if it is excessively difficult to add new functionality or make other changes.

Fourth, software that is decaying may nevertheless be increasing in value. Indeed, the very changes that "cause" decay also increase the value of the software.

Fifth, implicit in our definition is the idea that code decay is the result of previous changes to the software.[3] Thus, there are "actionable" means to prevent, retard or remediate code decay. The "no decay without change" concept, however, operates only at a high level. That a region of the code can decay as the result of changes elsewhere is entirely possible.

Finally, the "harder to change than it should be" aspect of code decay, while central, is also elusive. Some code is simply inherently hard to change, and to attribute this to decay is misleading. Many of the code decay indices in §IV adjust for this by means of scaling, for either the size of code units or time. In addition, difficulty of change is a function of the developer making the change. A definitive adjustment for developer ability has not been devised, and usually we must relegate developer variability to "noise" terms in our models.

B. Causes of Code Decay

In a sense, change to code is the cause of decay. As change is necessary to continue increasing the value of the software, a useful concept of a cause must allow it to be present or absent in a project under active development. Causes of decay reflect the nature of the software itself as well as the organizational milieu within which it is embedded. Examples include:[4]

1. *Inappropriate architecture* that does not support the changes or abstractions required of the system.
2. *Violations of the original design principles*, which can force unanticipated changes to violate the original system assumptions. Changes that match the original design tend to be comparatively easy, while violations not only are difficult to implement, but also can lead future changes to be difficult

[3] That is, there is no "natural" or "physical" decay.

[4] There is no implication every cause is present in any given situation.

as well. In switching systems, for example, many of the original system abstractions assume that subscriber phones remain in fixed locations. The changes required to support wireless phones that roam among cell sites were unanticipated by the original system designers. Note that this cause can be difficult to distinguish from inappropriate architecture.

3. *Imprecise requirements*, which can prevent programmers from producing crisp code, causing developers to make excessive numbers of changes.
4. *Time pressure*, which can lead programmers to take shortcuts, write sloppy code, use kludges (see §III-C), or make changes without understanding fully their impact on the surrounding system.
5. *Inadequate programming tools*, for example, unavailability of computer-aided software engineering (CASE) tools.
6. *Organizational environment*, manifested, for instance, in low morale, excessive turnover or inadequate communication among developers, all of which can produce frustration and sloppy work.
7. *Programmer variability*, for example, programmers who cannot understand or change delicate, complex code written by their more skilled colleagues.
8. *Inadequate change processes,* such as lack of a version control system or inability to handle parallel changes [15]. (This cause is particularly pertinent to today's world of Web distribution of open source software.)

Bad project management may amplify the effects of any of these causes.

C. Symptoms of Code Decay

In our conceptual model, symptoms are measurable manifestations of decay, in the same way that chest pains are a symptom of heart disease. Some of the code decay indices in §IV are measurements of symptoms.

Below we list plausible symptoms of decay.

1. *Excessively complex (bloated)* code is more complicated than it needs to be to accomplish its task. If rewritten, bloated code could become easier to understand and simpler to maintain. Standard software "metrics" are potential means to measure complexity, but they are designed to measure de facto complexity, rather than the difference between de facto and inherent complexity. Based on discussions with developers, one promising candidate is *nesting complexity*: the nesting complexity of a line of code is the number of loops and conditionals enclosing it.[5] An alternative form of complexity, which is especially troublesome to developers, is treated in item 6 below.
2. *A history of frequent changes*, also known as code churn, suggests prior repairs and modifications. If change is inherently risky, then churn signifies decay.
3. Similarly, code with a *history of faults* may be decayed, not only because of having been changed frequently, but also because fault fixes may not represent the highest quality programming.[6]
4. *Widely dispersed changes* are a symptom of decay because changes to well-engineered, modularized code are local. As discussed in §V-A, this symptom produces clear scientific evidence of code decay.
5. *Kludges* in code occur when developers knowingly make changes that could have been done more elegantly or efficiently.[7] While not an "official" categorization, kludged code is often identified literally as such in MR abstracts. That kludged code will be difficult to change is almost axiomatic.
6. *Numerous interfaces* (for example, entry points) are cited frequently by developers when they describe their intuitive definition of code decay. As the number of interfaces increases, increasing attention must be directed to possible side-effects of changes in other sections in the code.

D. Risk Factors for Code Decay

Risk factors, as in medicine, increase the likelihood of code decay, or exacerbate its effect. By themselves, they are not necessarily indicators or causes of decay, but are cause for concern even in

[5]Nesting complexity would capture the addition of features to the system by means of conditionals.
[6]Of course, fault-prone code may also simply be inherently complicated.
[7]For example, in response to schedule pressure.

the absence of symptoms.

1. *Size.* The size of a module m, in our analyses best measured by NCSL(m), the number of non-commentary source lines,[8] is clear cause for concern. The larger the module, the more likely essentially any of the symptoms in §III-C is present.
2. *Age* of code is a clear risk factor, but intuition regarding age is complicated. On the one hand, aged code may be a risk factor for decay if the code is neglected, or simply because older code units have had more opportunity to be changed, and their original environment is less likely to have persisted. On the other hand, code that is so stable that no change is necessary may not be decayed at all. Indeed, because of conflicting pressures, variability of age within a code unit may be the essential characteristic.
3. *Inherent complexity* is a risk factor for decay despite our defining code decay in a manner that adjusts for complexity ("harder to change than it should be"). Inherent complexity is also relevant when comparing one system to another: because it is inherently more complex, real-time software is more likely to decay than standard MIS applications.
4. *Organizational churn* (for example, turnover or reorganization) increases the risk of decay by degrading the knowledge base, and can also increase the likelihood of inexperienced developers changing the code (see item 7). Organizational churn is not readily discerned from the version management database; however, a parallel organizational study, reported in part in [16], links decay to events in the history of AT&T and Lucent.
5. *Ported code* was originally written in a different language, for a different system, or for another hardware platform. Both the porting process itself and the new milieu are risks for decay.
6. *Requirements load*, when heavy, means that the code has extensive functionality and is subject to many constraints. Multiple requirements are hard to understand, and the associated functionality is hard to implement, resulting in a higher risk of decay. In addition, a heavy requirement load is likely to have accreted over time, so that the code is doing things it was not designed to do.
7. *Inexperienced developers* can be either new to programming or new to particular code. They increase the risk of decay because of lack of knowledge, a lack of understanding of the system architecture, and (for those early in their careers) potential for lower or less-developed skills.

IV. Code Decay Indices

In the software engineering literature there is a rich history of studies involving software measurement and measurement theory (see, for example, [17]). Our code decay indices follow in this tradition, by being both *quantified* and *observable in the version management data base.* Pursuing the medical metaphor, CDIs may be interpreted as quantified symptoms, quantified risk factors, or prognoses, which are predictors of the responses (cost, interval, quality). Ordinarily, prognoses are functions of quantified symptoms and risk factors.

In order to define actionable priorities to remediate decay, indices must encapsulate developer knowledge and discriminate over both time and location in the software. Also, several of the indices can be visualized in compelling ways, as we illustrate in §IV-B.4 and §V-B.

A. General Considerations

When defining a CDI, one confronts three critical issues.

The first issue is to select appropriate levels of *aggregation* for both changes and software units. Of the levels of changes described in §II, MRs seem in most instances to be the most informative: the associated data sets are rich enough to be interesting, but not so large as to create intractability.

For most of the system we study, software can be aggregated to any of three levels.[9] Files are the atomic unit of software. Modules are collections of related files, corresponding physically to a single

[8]Obtained by summing over all files f belonging to m.

[9]Lines, even though tracked in the version management data base, simply lack sufficient structure to be appropriate.

directory in the software hierarchy. A subsystem is a collection of modules implementing a major function of the software system. In our studies, modules typically yield the most insight.

The second issue is *scaling:* in some cases it is helpful to scale a CDI to convert it into a rate per unit time or per unit of software size (usually, NCSL, the number of non-comment source lines). In addition to being scaled for time, indices may also be functions of time, in order to illuminate the evolution of code decay.

The third issue is *transformation:* an index can sometimes be improved by transforming a variable mathematically, for example, by taking logarithms, powers or roots. In some cases, the rationale may be physical, while in others it will be statistical, in order to improve the "fit" of models.

B. Example CDIs

Here, we present example CDIs that appear in the analyses in §V. They represent symptoms, risk factors and prognoses of decay. Candidates for other symptoms and risk factors will be presented in future papers.

We use the following notation: c denotes changes (as noted above, most often MRs); ℓ denotes lines of code; f denotes files; m denotes modules; d denotes developers. None of these objects is subscripted, so that (for example) $\sum_c$ denotes a sum over all changes.

We employ $|S|$ to denote the number of elements in a set S, and for a change c and software unit m, $c \rightsquigarrow m$ means that "c touches m:" some part of m is changed by c. Also $\mathbf{1}\{\mathbf{A}\}$, the indicator of the event A, is equal to one if A occurs and zero otherwise.

In addition, several of the CDIs (all computable directly from the version management data base) depend on characteristics of changes:

$$\begin{aligned}
\mathrm{FILES(c)} &= \sum_f \mathbf{1}\{\mathbf{c} \rightsquigarrow \mathbf{f}\}, \text{the number of files touched by } c \\
\mathrm{DELTAS(c)} &= \text{number of deltas associated with } c \\
\mathrm{ADD(c)} &= \text{number of lines added by } c \\
\mathrm{DEL(c)} &= \text{number of lines deleted by } c \\
\mathrm{DATE(c)} &= \text{the date on which } c \text{ is completed, which we term just the date of } c \\
\mathrm{INT(c)} &= \text{the } \textit{interval} \text{ of c, the (calendar) time required to implement } c \\
\mathrm{DEV(c)} &= \text{number of developers implementing } c.
\end{aligned}$$

B.1 History of Frequent Changes

The historical count of changes is expressed by the CDI

$$\mathrm{CHNG(m,I)} = \sum_{\mathrm{c} \rightsquigarrow \mathrm{m}} \mathbf{1}\{\mathrm{DATE(c)} \in \mathrm{I}\}, \tag{1}$$

the number of changes to a module m in the time interval I, appears in §V-B. In other settings, the frequency of changes may be more relevant, as quantified by

$$\mathrm{FREQ(m,I)} = \frac{1}{|\mathrm{I}|} \sum_{\mathrm{c} \rightsquigarrow \mathrm{m}} \mathbf{1}\{\mathrm{DATE(c)} \in \mathrm{I}\}. \tag{2}$$

B.2 Span of Changes

The *span* of a change is the number of files it touches (Here files yield a more sensitive index than modules.), leading to the CDI

$$\mathrm{FILES(c)} = \sum_{\mathrm{f}} \mathbf{1}\{\mathbf{c} \rightsquigarrow \mathbf{f}\}. \tag{3}$$

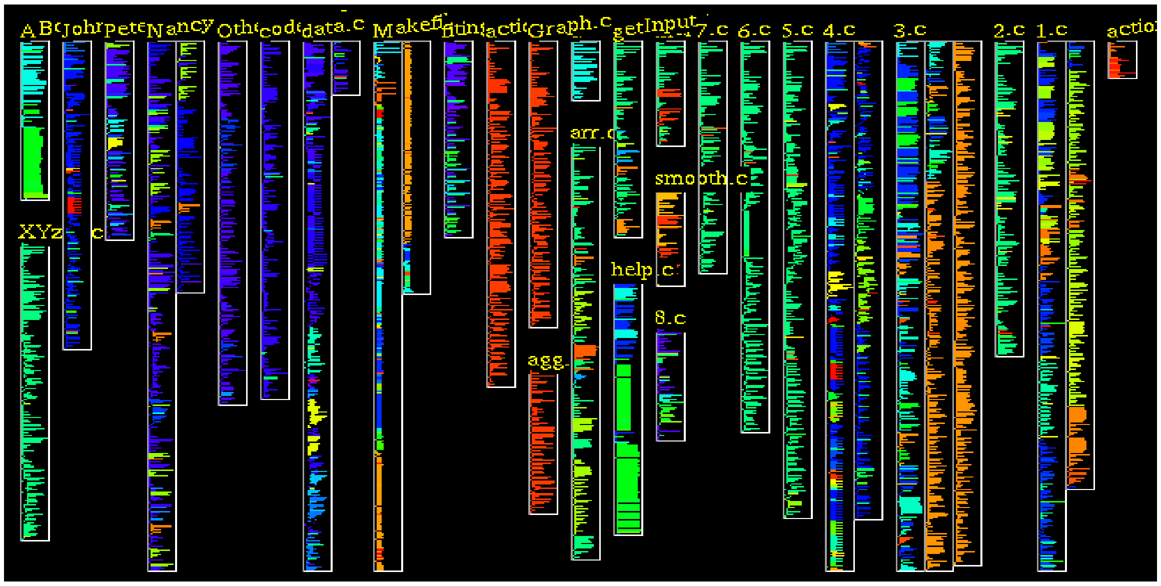

Fig. 2. The SeeSoft view of one module. Color represents the age of a source code line. Rainbow colored boxes represent frequently changed files, while boxes with a single hue represent files that changed little since their creation.

In §V-D we will provide evidence that FILES(·) predicts the effort necessary to make changes. There are three primary reasons why changes touching more files are more difficult to accomplish, and hence that span is a symptom of decay. First is the necessity to get expertise about unfamiliar files from other developers; this is especially vexing in large-scale software, where each developer has a localized knowledge of the code. Second is the breakdown of encapsulation and modularity. Well-engineered code is modular and changes are localized. Changes spanning multiple files are more likely to modify an interface. Third is the size: touching multiple files significantly increases the size of the change.

In the subsystem we study (§V), FILES(·) increases over time, so that this CDI produces clear scientific, symptomatic evidence of code decay, as discussed in §V-A.

B.3 Size

The size of a module m has already been defined as NCSL(m), the number of non-commentary source lines, obtained by summing over all files f belonging to m. Although we do not elaborate in §V, extensive analyses show most standard software "complexity" metrics ([18]) are nearly perfectly correlated with NCSL, so that size is effectively synonymous with complexity.

B.4 Age

We define the age of a software unit as the average age of its constituent lines. For a module m, this is given by

$$\mathrm{AGE(m)} = \frac{1}{|\mathrm{m}|} \sum_{\mathrm{f} \in \mathrm{m}} \sum_{\ell \in \mathrm{f}} \mathrm{AGE}(\ell), \tag{4}$$

where $f \in m$ means f is a file in module m and $\ell \in f$ means ℓ is a line in the file f. Use of AGE in a predictive CDI is illustrated in §IV-B.5.

Also interesting is the variability of the ages of the lines in a code unit, The SeeSoft view [19] in Figure 2 shows the variability of age in one module. The files are represented by boxes (labels have been changed for confidentiality) and the source code lines are represented by colored lines within the boxes. The color of a line represents its age: files in which age is highly variable stand out with most of the rainbow colors. The files that changed little since their creation contain mostly a single hue.

B.5 Fault Potential

Predictive CDIs are functions of CDIs that quantify symptoms or risk factors, and are intended to predict the key responses of effort, interval and quality. We present three such indices, two dealing with quality and one with effort, which are discussed at more length in §V-C–V-D.

Predictors of the number of faults that will have to be fixed in module m (and, thus, of the quality response) in a future interval of time, taken from [20], include the weighted time damp model

$$\mathrm{FPOT_{WTD}(m,t) = a \sum_{c \rightsquigarrow m, DATE(c)<t} e^{-\alpha[t-DATE(c)]} \log\left[ADD(c,m) + DEL(c,m)\right]}, \tag{5}$$

where a and α are determined by statistical analysis, and the generalized linear model

$$\mathrm{FPOT_{GLM}(m,t) = a \times \sum_{c \in \Delta} \mathbf{1}\left\{\mathbf{c} \rightsquigarrow \mathbf{m}\right\} \times \mathbf{b}^{AGE(m)}}, \tag{6}$$

where Δ is the entire set of deltas[10] up to time t and AGE is given by (4).

Both of these indices illuminate change as the primary agent creating faults (Even though faults do not arise spontaneously, this is not a tautology: the absence of other terms such as size and complexity is highly informative.), but depict differing temporal effects. In (5), the effects of changes are "damped" and attenuate over time, while in (6) faults are less likely in older code (provided b is estimated to be less than one, as in our data). Statistical analyses of the models appear in §V-C.

B.6 Effort

A predictor of the effort (person-hours) required to implement a change is

$$\begin{aligned} \mathrm{EFFORT(c)} \quad = \quad & a_0 + a_1\mathrm{FILES(c)} + \mathrm{a_2} \sum_{\mathrm{f}} \mathbf{1}\left\{c \text{ touches } f\right\} |\mathbf{f}| \\ & + a_3\mathrm{ADD(c)} + \mathrm{a_4DEL(c)} + \mathrm{a_5INT(c)} + \mathrm{a_6DEV(c)}. \end{aligned} \tag{7}$$

One motivation for (7) is to distinguish the *dependency overhead* associated with a change — captured in the terms involving a_0, a_1 and a_2 — from the *nominal effort*, represented by the terms involving a_3 and a_4. The remaining terms incorporate interval and *developer overhead*. A statistical analysis of this index appears in §V-D.

V. The Evidence for Decay

In this section, we discuss some of our major results to date. All of these analyses are based on a single subsystem of the code, consisting of approximately 100 modules and 2500 files. The change data consist of roughly 6000 IMRs, 27,000 MRs and 130,000 deltas. Some 500 different login names made changes to the code in this subsystem.

The results yield very strong evidence that code does decay. First, in §V-A, statistical smoothing demonstrates that the span of changes (see (3)) increases over time, which is a clear symptom of code decay. This analysis is extended, in §V-B, by network-based visualizations showing that the increase in span is accompanied by (and may cause) a breakdown in the modularity of the code.

Our other results show how decay affects two of the three key responses, namely quality and effort. In §V-C, we present models involving the fault potential CDIs of §IV-B.5. Finally, in §V-D, we present a statistically estimated version of the CDI EFFORT in (7), together with some intriguing implications, including indications that changes with large spans tend to require large efforts. This underscores the importance of the preceding sections that demonstrate increasing span of changes.

Not all of the evidence is conclusive or complete, and in some cases, multiple interpretations are possible. For example, some of the increase in span of changes (§V-A) and decrease in modularity

[10]Here changes are deltas rather than MRs.

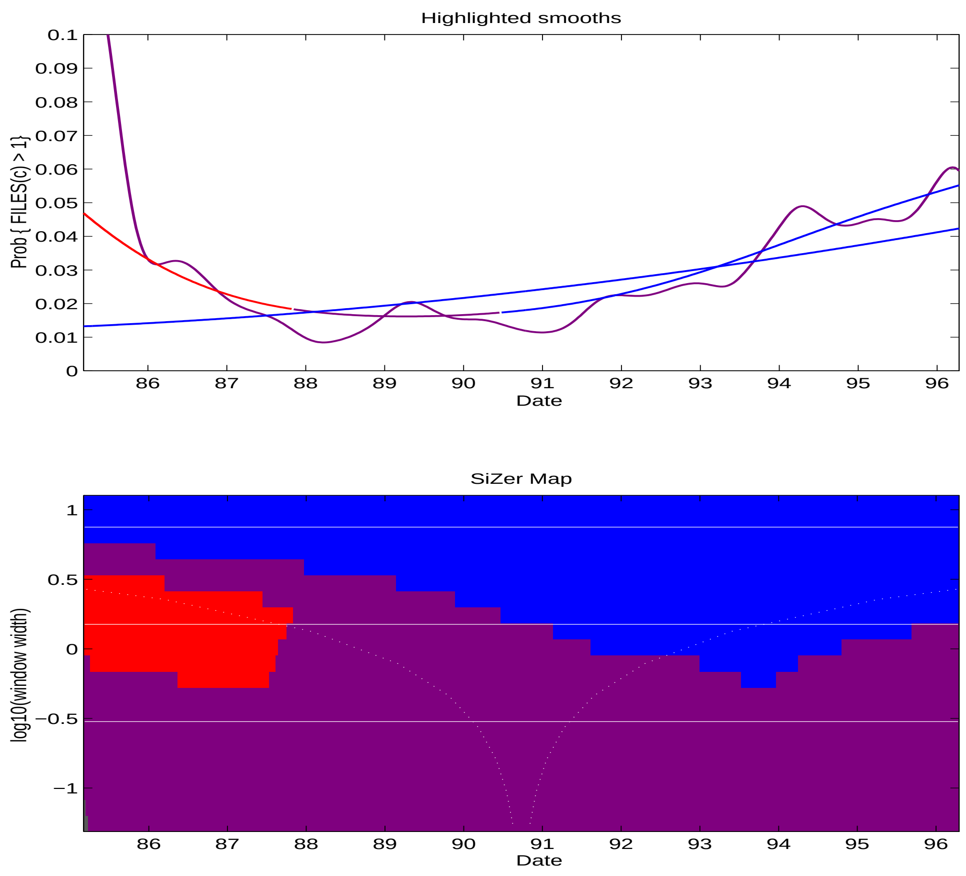

Fig. 3. SiZer maps of numbers of files touched by changes through time. The overall trend has been a significant increase in the difficulty of changes. At a finer resolution, there was a decreasing trend during the developmental phase of the subsystem, when changes likely involved multiple files, but this trend reversed before long.

(§V-B) can be attributed merely to growth of the subsystem. Similarly, the fault potential analysis in §V-C, identifies change as a causal agent for faults, but does not differentiate decay among modules.

Collectively the results show that our change-based conceptual model of decay is the right one. That change is the agent of decay is crucial is borne out by the data, which is crucial since there are then actionable means to retard or reverse decay.

A. Temporal Behavior of the Span of Changes

The CDI FILES(c) of (3) measures the difficulty of a change by how many code units files need to be changed in order to implement it. An increase in the span of changes, then, is symptomatic of decay, as discussed in § IV-B.2.

Figure 3 shows that span is increasing for the subsystem under study. There, we display the chance that at any given time an MR touches more than one file by smoothing data in which each point corresponds to an MR. A point's x-coordinate is time represented its the opening date, and its y-coordinate is one when more than one file is touched, and zero otherwise. Three local linear smooths (See, e.g., [21] and [22] for introduction and discussion.) are shown in the top plot. These smooths are essentially weighted local averages, where the weights have a Gaussian shape, and the widths of the windows (i.e., standard deviation of the weight function) are $h = 0.3$ (purple curve), $h = 1.5$ (multicolored curve) and $h = 7.5$ (blue curve).

The central curve, $h = 1.5$, shows an initial downward trend, which is natural because many files are touched by common changes in the initial development phase, followed by a steady upward trend starting in 1990. This last trend reflects breakdown in the modularity of the code, as we discuss further in §V-B. That this is a substantial increase comes from the fact that values on the y-axis represent probabilities (local in time) that a change will touch more than one file, which more than double from a low of less than 2% in 1989 to more than 5% in 1996.

In the absence of more detailed analysis, the results in the top plot in Figure 3 depend on the window width h. The larger window width $h = 7.5$ shows only the upward trend, while the smaller window width $h = 0.3$ shows a lot of additional structure, which may be "microtrends" or may instead be spurious sampling artifacts. But how can we be sure? Furthermore, how do we know the features observed in the $h = 1.5$ smooth, which contains the important lessons, are real?

The bottom half of Figure 3 is a *SiZer map*,[11] which addresses this issue. Each location corresponds to a date, and also to a window width h, and is shaded blue (red) when the smooth at that window width and date is (statistically) significantly increasing (decreasing, respectively). Regions where there is no significant change are shaded in the intermediate color purple.

The smallest window width, $h = 0.3$, in the top plot is represented by the bottom white line in the lower plot, and is shaded purple in the top plot, since this window width is shaded purple at all dates in the SiZer map. This is interpreted as "when the data are studied at this level of resolution, there are no significant increases or decreases," i.e., the wiggles in the curve are not statistically significant.

The intermediate window width $h = 1.5$ runs through both the red and blue shaded regions. This same coloring is used in the curve in the top plot, which shows that the structure is statistically significant. In particular there is an important downward trend at the beginning, and upward trend after 1990.

The large window width $h = 7.5$ runs through the region that is shaded entirely blue in the bottom plot, and thus inherits this color in the top plot. This shows that when the data are smoothed nearly to the point of a doing a simple linear least squares fit, the resulting line slopes significantly upwards.

These conclusions are complementary rather than inconsistent, because SiZer shows what is happening *at each scale of resolution.* When the data are not smoothed too much, there can be a decrease in one region, which when the data are smoothed very strongly, becomes overwhelmed by the increases elsewhere.

B. Time Behavior of Modularity

A key tenet of modern programming practice is modularity: code functionality should be local, so that changes will be also. In the system we analyzed, subsystems are divided into modules by functionality, and this division is successful to the extent to which when working on one module a developer need not devote significant attention to the effects on other modules. Conversely, changes that require modifications of many modules are likely to be more difficult to make correctly.

Alone, the increase in span of changes described in §V-A does not imply breakdown of the modularity of the subsystem. Some increase in span could reflect simply the growth of the subsystem, and even changes of wider span need not cross module boundaries. The network visualization tool NicheWorks [24] allows us to address the question of whether modularity is breaking down over time, and leads to the results in Figure 4, which suggest strongly that it is.

Each diamond-shaped icon in the upper left panel of Figure 4 corresponds to a module; the positions of the modules have been chosen by NicheWorks in a manner that places pairs of modules nearby if they have been changed together as part of the same MRs a large number of times. More precisely, the weights are defined in terms of the "number of changes" CDI of (1), with that for modules m and m' being

$$w(m, m') = \frac{\mathrm{CHNG(m, m', I)}}{\sqrt[4]{\mathrm{CHNG(m, I)} \times \mathrm{CHNG(m', I)}}}, \tag{8}$$

[11]SiZer, its properties, and some variations are discussed in [23].

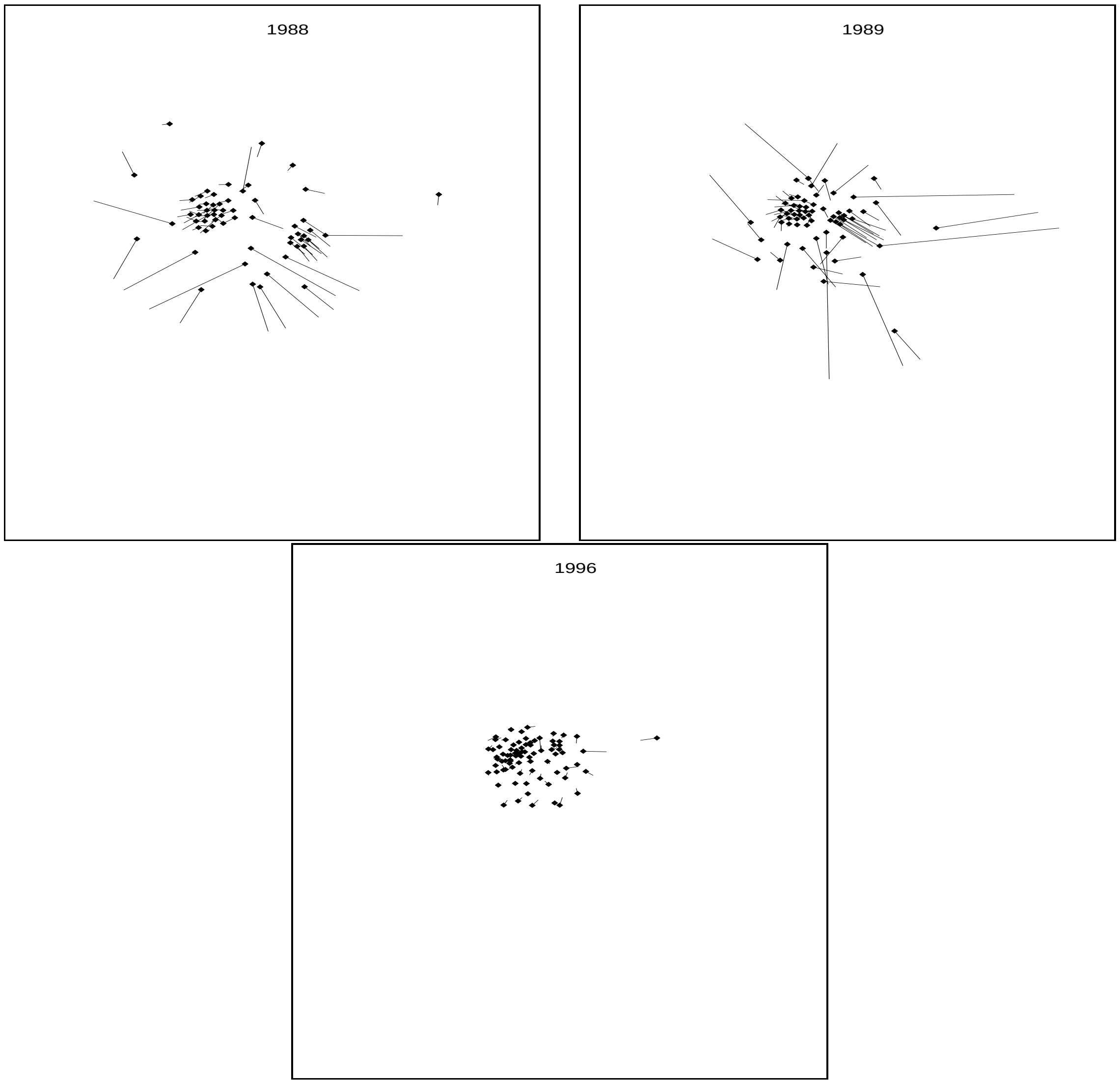

Fig. 4. Top left: NicheWorks view of the modules in one subsystem, using change data through 1988 to place modules which have been changed at the same times close to one another. Two clusters of modules are evident; a module within one of these clusters is often changed together with other modules in the cluster but not with other modules. Top right: NicheWorks view of the modules in the top left, this time incorporating the change history through 1989. The clusters that appear in the top left view are converging in on each other. This suggests that the architecture that was previously successful in separating the functionality of the two clusters of modules is breaking down. Bottom: the breakdown continued, and at the end of 1996, there was no suggestion of multiple clusters of modules.

where I is an interval of time (see below) and where

$$\mathrm{CHNG}(\mathrm{m},\mathrm{m}',\mathrm{I}) = \sum_{\mathrm{c} \rightsquigarrow \mathrm{m}, \mathrm{c} \rightsquigarrow \mathrm{m}'} \mathbf{1}\left\{\mathrm{DATE}(\mathrm{c}) \in \mathrm{I}\right\}$$

is the number of MRs touching both m and m'. In the upper left panel, the diamonds show this network view using all the change data through the end of calendar year 1988 (corresponding to one choice of I), while the other ends of the segments connected to the diamonds display the same view at the end of 1987 (an earlier choice of I).

In this way, one can see how relationships among modules evolved through time. In the top left panel of Figure 4, there are two main clusters of roughly a dozen modules each. In the top right panel, however, which displays the change data through the end of 1989 in the locations of the diamonds (the 1988 data appear here as the locations of the segments' endpoints), these two clusters have mostly merged. The merging process continued, and at the end of 1996, the clusters are no longer visible. While the logic of the code was originally intended to see to it that some modules would be essentially independent of each other, new and unanticipated functionality may have helped to destroy this independence.

The weights in (8) constitute a compromise between simple counts

$$w(m, m') = \mathrm{CHNG(m, m', I)},$$

which tend to place too close together pairs of modules that are touched together frequently only because they are touched large numbers of times in total, and

$$w(m, m') = \frac{\mathrm{CHNG(m, m', I)}}{\sqrt{\mathrm{CHNG(m, I)} \times \mathrm{CHNG(m', I)}}}, \tag{9}$$

which can be interpreted as a correlation (It is dimensionless and lies between zero and one.), but which can exaggerate relationships between modules that are rarely touched at all.

One shortcoming is that the weights of (8) (unlike those of (9)) are not invariant with numbers of changes. The following dispersion analysis, however, provides further evidence of the decline of modularity. In 1988, the mean square distance between points in the small, eleven-point cluster and its centroid is 0.355, while the average distance between points in the larger, 26-point cluster and its centroid is 0.526. The inter-cluster distance, or the distance between the centroids of the two clusters, is 2.78. An intuitively appealing measure of distance between clusters, then, is $2.78/\sqrt{0.355 \times 0.526} = 6.43$. The analogous quantity for 1989 is $1.35/\sqrt{0.306 \times 0.419} = 3.77$. After the large decrease in 1989, this measurement continues to shrink, albeit not as rapidly, reaching $1.40/\sqrt{0.330 \times 0.469} = 3.56$ in 1996.

C. Prediction of Faults

In §III-A, we identified quality as one of three key responses to code decay. Here we summarize research linking faults in the software to symptoms of code decay, using the predictive CDIs $\mathrm{FPOT_{WTD}}$ of (5) and $\mathrm{FPOT_{GLM}}$ of (6). More complete discussion of this fault potential modeling appears in [20].

The thrust of these models is to predict the distribution of future faults over modules in the subsystem from the modules' change history. The best models predicted numbers of faults using numbers of changes to the module in the past, the dates of these changes (*i.e.* the negative of their ages, measured in years), and their sizes, as in (5):

$$\mathrm{FPOT_{WTD}(m)} \propto \sum_{\mathrm{c} \rightsquigarrow \mathrm{m}} \mathrm{e}^{0.75 \times \mathrm{DATE(c)}} \log\left[\mathrm{ADD(c, m)} + \mathrm{DEL(c, m)}\right], \tag{10}$$

with the parameter $\alpha = .75$ determined by statistical analysis (see [20]). Thus, large, recent changes add the most to fault potential, and the number of times a module has been changed is a better predictor than its size of the number of faults it will suffer in the future. That $\alpha \neq 0$ is the primary (and direct) evidence that changes induce faults: were $\alpha = 0$, past changes of the same size would be indistinguishable from one another, and hence none could be posited to have any specific effect.

The model (10) does provide evidence that some modules are more decayed than others. In principle, this issue could be addressed by allowing α to be module dependent, but we have not yet done this.

An alternative (and less powerful) model, using the CDI of (6) and the same data as (10), is a generalized linear model, is

$$\mathrm{FPOT_{GLM}(m)} = .017 \times \sum_{\mathrm{c}} \mathbf{1}\left\{\mathbf{c} \rightsquigarrow \mathbf{m}\right\} \times \mathbf{.64}^{\mathrm{AGE(m)}}. \tag{11}$$

This model implies that code having many lines that have survived for a long time is likely to be relatively free of faults. More precisely, according to (11), code a year older than otherwise similar code tends to have only two-thirds as many faults.

One way to evaluate these models is by comparison with a "naive" model that predicts the number of future faults in given locations to be proportional to the number of past faults. As discussed in [20], in some cases, (11) is only marginally superior to the naive model (as measured by a Poisson deviance). Nevertheless, this still means that a model suggesting causality (deltas cause faults) has the same explanatory power as a model positing simply that the distribution of faults over modules is stationary over time.

Simulations of deviances provide strong evidence that the model (10) is superior to that of (11). In particular, this means that treating changes individually improves the predictions.

Equally important is that other variables did not improve the predictions, once size and time of changes are taken into account. In particular, predictions do not improve by including either module size or other measures (metrics) of software complexity (which in our data are correlated essentially completely with size). Thus, changes to code are more responsible for faults than the complexity of the code.

Moreover, the number of developers touching a module had no effect on its fault potential.[12] One possible explanation is that strong organization programming standards attenuate any such effects. The change from code ownership to change ownership ([16]) is a confounding factor in this regard.

Finally, concurrent changes with large numbers of other modules did not contribute to fault potential. In one sense, this suggests that the decline of modularity described in §V-B may not be harmful, but since the size of changes is correlated with their span, it is more likely that we are simply seeing the size variable mask the effect of span.

D. *Models for Effort*

Here we assess the evidence for "bottom line" relevance of code decay: can the effort required to implement changes be predicted from symptoms and risk factors for decay? The analysis employs a variant of the predictive CDI EFFORT of (7), with the "sum of touched file sizes" term in (7) omitted. The results are suggestive but, because of the small sample size, not definitive.

The model was fit using data from a set of 54 *features*. As noted in §II-A, features are the units of system functionality (e.g., call waiting) by which the system is extended, and are too aggregated for most purposes. However, effort data (person hours) are available only at this level. (Imputation of disaggregated effort is addressed in [25].)

Extreme variability of the feature-level data necessitated taking logarithms of all variables. (The actual transformation — $\log[1+\cdot]$ — avoids negative numbers.) The resultant model is

$$\begin{aligned}\log(1+\mathrm{EFFORT(c)}) \;=\;& .32 + .13 \times (\log[1+\mathrm{FILES(c)}])^2 \\ & -.09 \times (\log[1+\mathrm{DEL(c)}])^2 \\ & +.12 \times \log[1+\mathrm{ADD(c)}] \times \log[1+\mathrm{DEL(c)}] \\ & +.11 \times \log[1+\mathrm{INT(c)}] - .47 \times \log[1+\mathrm{DELTAS(c)}]\end{aligned} \tag{12}$$

All coefficients shown are statistically significantly different from zero; the multiple R^2 value is .38. Despite the danger that this model is "overfit," removal of any of the variables decreases the fit dramatically.

Some interpretations of (12) seem clear. First, dependence on FILES(c) confirms that the span of changes is indeed a symptom of decay; that the dependence is quadratic hints that moderate span

[12]One might expect that modules modified by many developers would have confused logic as a result of the different styles, and hence be difficult to change.

may not be serious. Second, as hypothesized in §IV-B.6, dependency overhead (in (12), embodied in EFFORT(c)) can be distinguished from nominal effort (terms involving ADD(c) and DEL(c)).

Other interpretations seem more problematic. For example, the negative coefficient for $(\log[1 + \mathrm{DEL(c)}])^2$ deletions are accomplished relatively quickly (which makes sense), but can also be interpreted as simply fitting cases in the data where large numbers of lines are deleted. Similarly, the interaction term between additions and deletions ($\log[1 + \mathrm{ADD(c)}] \times \log[1 + \mathrm{DEL(c)}]$) may suggest that the hardest changes are those requiring both additions and deletions, but the high level of aggregation mandates caution when trying to extrapolate this to, say, the delta level.

The negative coefficient for $\log[1+\mathrm{DELTAS(c)}]$ is puzzling, since it is difficult to believe that features containing large numbers of editing changes are somehow easier to implement. But removing this single term decreases R^2 nearly by one-half, so there is no doubt that the effect is present in the data. Detailed examination of the data suggests that the negative coefficient is picking up the approximately 5 cases (10% of the data) in which effort is large (close to the maximum) effort, but the number of deltas is very small.

E. Confirmatory Evidence

The results reported in this paper are derived primarily from statistical analysis of change management data. They are corroborated by results reported in [16], which is part of the same code decay project.

VI. Discussion

Using tools developed to handle change management data, a conceptual model of code decay (associated concepts of causes, symptoms and risk factors), code decay indices and statistical analyses, we have found evidence of decay in the software for a large telecommunications system.

Four specific analyses were performed. They demonstrate (1) Increase over time in the number of files touched per change to the code; (2) The decline in modularity of a subsystem of the code, as measured by changes touching multiple modules; (3) Contributions of several factors (notably, frequency and recency of change) to fault rates in modules of the code; and (4) That span and size of changes are important predictors (at the feature level) of the effort to implement a change.

At the same time, evidence of dramatic, widespread decay is lacking. Retrospectively, this is not surprising: the system studied is a fifteen-year old, successful product to which new features can still be added.

The tools, concepts and analyses are transferable to any software project for which comparable change management data exist. We anticipate that all projects of sufficiently large scale will exhibit decay to some extent: that is, code decay is a generic phenomenon.

Current investigations are focusing on the effectiveness and economic efficiency of means to prevent or retard code decay, such as perfective maintenance. Whether (in the medical metaphor), code decay can ultimately be fatal is not clear. There are, however, anecdotal reports of systems that have reached a state from which further change is not possible.

Acknowledgments

We gratefully acknowledge conversations and contributions from Mark Ardis, David Parnas, Harvey Siy, Lawrence Votta, and David Weiss.

References

[1] A. Mockus, S. G. Eick, T. L. Graves, and A. F. Karr, "New roles for change management data in software engineering," *Technical Report, National Institute of Statistical Sciences*, 1999.
[2] L. A. Belady and M. M. Lehman, "Programming system dynamics, or the meta-dynamics of systems in maintenance and growth," Tech. Rep., IBM Thomas J. Watson Research Center, 1971.
[3] L. A. Belady and M. M. Lehman, "A model of large program development," *IBM Systems Journal*, pp. 225–252, 1976.
[4] M. M. Lehman and L. A. Belady, *Program Evolution: Processes of Software Change*, Academic Press, 1985.

[5] I. Vessey and R. Weber, "Some factors affecting program repair maintenance: An empirical study," *Communications of the ACM*, vol. 26, pp. 128–134, 1983.

[6] S. Bendifallah and W. Scacchi, "Understanding software maintenance work," *IEEE Transactions on Software Engineering*, vol. 24, pp. 311–323, 1987.

[7] W. S. Scacchi, "Managing software engineering projects: A social analysis," *Transactions on Software Engineering*, vol. 10, no. 1, pp. 49–59, January 1984.

[8] C. F. Kemerer and A. K. Ream, "Empirical research on software maintenance: 1981–1990," Tech. Rep., Massachusetts Institute of Technology, 1992.

[9] D. L. Parnas, "Software aging," in *Proceedings 16th International Conference On Software Engineering*, Los Alamitos, California, 16 May 1994, pp. 279–287, IEEE Computer Society Press.

[10] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 886–894, December 1996.

[11] V. R. Basili and B. T. Perricone, "Software errors and complexity: An empirical investigation," *Communications of the ACM*, vol. 27, no. 1, pp. 42–52, January 1984.

[12] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *IEEE Transactions on Software Engineering*, vol. 10, no. 6, pp. 728–737, 1984.

[13] E. B. Swanson, "The dimensions of maintenance," in *Proc. 2nd Conf. on Software Engineering*, San Francisco, 1976, pp. 492–497.

[14] K. H. An, D. A. Gustafson, and A. C. Melton, "A model for software maintenance," in *Proceedings of the Conference in Software Maintenance*, Austin, Texas, September 1987, pp. 57–62.

[15] D. E. Perry, H. P. Siy, and L. G. Votta, "Parallel changes in large scale software development: An observational case study," in *Proceedings of the 1998 International Conference on Software Engineering*, Kyoto, Japan, April 1998.

[16] N. Staudenmayer, T. L. Graves, J. S. Marron, A. Mockus, H. Siy, L. G. Votta, and D. E. Perry, "Adapting to a new environment: how a legacy software organization copes with volatility and change," in *5th International Product Development Conference*, Como, Italy, 1998.

[17] S. L. Pfleeger, R. Jeffery, W. Curtis, and B. Kitchenham, "Status report on software measurement," *IEEE Software*, pp. 33–43, March/April 1997.

[18] H. Zuse, *Software Complexity: Measures and Methods*, de Gruyter, Berlin, New York, 1991.

[19] T. A. Ball and S. G. Eick, "Software visualization in the large," *IEEE Computer*, vol. 29, no. 4, pp. 33–43, April 1996.

[20] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, 1999, To appear.

[21] M. P. Wand and M. C. Jones, *Kernel Smoothing*, Chapman and Hall, London.

[22] J. Fan and I. Gijbels, *Local polynomial modelling and its applications*, Chapman and Hall, London, 1996.

[23] P. Chaudhuri and J. S. Marron, "Sizer for exploration of structures in curves," *Journal of the American Statistical Association*, 1999, To appear.

[24] G. J. Wills, "Nicheworks – interactive visualization of very large graphs," in *Graph Drawing '97 Conference Proceedings.* Springer-Verlag Lecture Notes in Computer Science, Rome, Italy, 1997.

[25] T. L. Graves and A. Mockus, "Inferring change effort from configuration management databases," *Metrics 98: Fifth International Symposium on Software Metrics*, November 1998.