An Empirical Exploration of Code Evolution

Alan Karr, National Institute of Statistical Sciences

Adam Porter*, University of Maryland

Lawrence Votta Jr., Lucent Technologies

1. Introduction

In the early 1980's, Belady and Lehman argued that successful software systems need to change frequently. To meet these needs, developers successively add needed functionality, even though this increases system size and complexity and becomes more and more difficult to do correctly.

Eventually this phenomenon -- we call it "code decay"-- makes it extremely, or even, prohibitively, expensive to add new functionality. In the most extreme cases costly redesign and redevelopment is the only way to prolong the system's lifetime.

Clearly, identifying, retarding, or preventing code decay would save considerable time and money throughout the software industry. Therefore, researchers have identified several possible causes and developed tools to defeat them. For instance some people argue that changes become difficult because, over time, developers lose their understanding of the system. Others believe that repeated changes erode the integrity of a system's internal structure. Still others think that some software designs are inherently less flexible than others.

1.1 Code Decay project

The degree to which any or all of these factors affect industrial systems is open to debate. Consequently, we are conducting a long-term, multidisciplinary project to discover the fundamental causes, symptoms, and remedies for code decay. The project team contains researchers in Statistics, Experimentation, Organizational Theory, Programming Languages, Software Engineering, and Visualization.

Our primary object of study is the AT&T 5ESS™ switching system. This includes the switch's source code and change control history covering the last 15years, its planned and actual development milestones, effort and testing data, organizational history, development policies, and coding standards.

Our goals are to (1) define response variables and document the existence of code decay, (2) develop code decay indices, (3) identify factors causing it, and (4) create and evaluate prevention strategies.

- **Responses**. Code decay may be very complex or occur only in certain circumstances and there may be several ways to measures it. (effort, total changes, fixes on fixes). Therefore, we will examine a variety of definitions for code decay, document its presence in our data, and develop precise measures (response variables) for it..
- **Indices.** An index is a surrogate measure used when a response variable is difficult to calculate. It can also serve as a predictor function when the response variable isn't well understood.
- **Factors.** After developing responses and indices, we will attempt to determine which factors inside the code or organization cause code decay.
- **Strategies.** Finally, we will attempt to identify and then evaluate strategies for preventing or retarding code decay.

1.2 Code decay testbed

The data we are using in this project are very rich. Yet with over 15 years of data to explore the likelihood of following blind alleys is very high. What we would like is a model of decayed systems that we can use to direct our research. Admittedly such models will be artificial and incomplete, but are important because

* This project is pat in part by a National Science Foundation Faculty Early Career Development Award, CCR-9501354, and by SBR-9529926.

they will help us to explore a wide variety of responses, indices, factors, and strategies, and quickly reject unpromising ones.
Below we will describe the code decay testbed, discuss the design used to create it, outline our initial analysis and results, and summarize its effect on the general project.

2. Accelerating Code Decay

One of the problems with analyzing industrial data is that we don't have any control over the data's creation or its contents. We can't, for example, control which changes are made, how they are implemented, nor who makes them. Therefore, it's sometimes difficult to interpret the data. Moreover since evolution can span many years, data collection is often limited due to cost considerations. To address these problems we developed a set of model systems called the code decay testbed.
Initially, the testbed contained one implementation for each of three systems. Each system's requirements are specified in temporal logic. We chose to use temporal logic because we had already developed a system that automatically constructs test harnesses and oracles and tests for violations of the temporal logic safety properties.
We also developed a set of five modification requests for each of the systems. We then asked several developers to implement the first modification request. After the modifications had passed acceptance test, we assigned the resulting implementations different developers and asked them to implement the next modification request. The process continued until all five modification requests were completed. Our hope is that the final implementations exhibit a high degree of code decay relative to the initial implementations.

2.1 Initial Systems.

The testbed contains 3 systems whose requirements are specified in temporal logic. Developers not involved in the study wrote the initial implementation using the C programming language.
• **Automatic Protection Switching System (APS):** manages redundant resources (working and protection lines) allocated to a telephone call to ensure that the highest quality lines are always in use (12 pages).
• **Traffic Signal Management System: (TRAFFIC):** controls traffic signals, turn arrows, and walk/don't walk displays at a multi-way intersection (8 pages).
• **Elevator Control System: (ELEVATOR):** controls the movement of a bank of elevators, their control panels, and emergency alarms in a multi-floor building (8 pages).

3. The Experiment

To create this testbed we designed and conducted the following experiment. Our goal was to create a set of code artifacts that exhibit some degree of code decay.

3.1 Experimental Setting

We ran this experiment during Spring 1996 at the University of Maryland. Sixteen graduate students in computer science acted as developers and the entire experiment took 6 weeks to complete.

3.2 Variables

This experiment manipulates several independent variables.
1. Conceptual integrity (the number of developers who modify an implementation.)
2. Implementation familiarity (the number of times a developer modifies a given implementation)
3. System Familiarity (the number of times a developer modifies any implementation of a given system)

Table 1 shows the experiment's design. For each modification we captured several dependent variables.
1. Self-reported effort.
2. The number of test runs required to pass acceptance test.
3. The testing results (i.e., specific safety properties violated)
4. The overall effect of the modification (i.e., some previously accepted safety properties now violated, no effect, some previously violated properties now accepted — no new violations).
5. Code metrics reflecting the system structure

		User ID															
		A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**
	Training	T0	E0	AO	T0	E0	AO	T0	E0	AO	T0	E0	AO	T0	E0	AO	TO
Change	**1**	E3	E1	A1	T2	E2	A2	E4	A4	T4	T1	T3	T5	A3	E5	A5	T6
ID	**2**	A3	E5	A1	E1	E2	A2	T3	E3	T4	A4	E4	T1	A5	T2	T5	T6
	3	A3	E5	A5	E1	A1	A2	E4	E3	T2	A4	T3	T1	T6	E2	T4	T5
	4	A3	E1	T3	E5	A1	A2	E3	T2	A5	T1	T5	A4	T6	E2	T4	E4

Table 1. This tables shows the design used in the experiment. Each subject began by modifying one of the intiial implementations (A0, E0, or T0). After the training round, there were 16 implementations (A1-A5, E1-E5, and T1-6).

3.3 Threats to validity

There are several threats to validity of this experiment

3.3.1 Threats to Internal Validity

Threats to internal validity are influences that can affect the dependent variables and mistakenly be considered the effect of an independent variable. We are aware of the following three threats.

- **Programmer behavior.** Different developers are likely to have different levels of programming skill and different development strategies.
- **System variances.** Some systems may be easier to understand or have better initial structure than others.
- **Change difficulty**. Within a given system some changes may be simpler to implement than others.

3.3.2 Threats to External Validity

Threats to external validity compromise our ability to generalize our results. We are aware of the following threats.

- **System size.** Our systems are very small compared to industrial systems. While they may be reasonable models for some types of changes, they cannot capture decay due to changing module relationships or architectures.
- **Subjects:** Our subjects are competent programmers, but may not be representative of professional programmers.
- **Time scale:** The passage of time or number of changes may be much less than that found in industrial systems..
- **Change complexity**. Our changes may not be representative of changes made to industrial systems.

3.4 Conducting the Experiment

We conducted the experiment in two phases: training and operation. In the training phase, we gave 6 hours of in-class instruction on temporal logic. We also provided 3 hours of instruction on the algorithms for converting temporal logic safety properties into test oracles. Finally we gave each student one of the initial implementations, its requirements specification and a modified specification that captured the intended change. The students then implemented the change. They used the testing tools, and had a 1-week deadline, exactly as they would in the actual experiment. After they completed the change one of the experimenters made himself available to answer any questions they encountered during the training activity.

Once a week for the following 4 weeks each student received a new implementation (generated in the previous week), all previous requirements specifications, and new specification detailing the intended change. They implemented the change within 1 week and we required all implementations to pass acceptance test before they were submitted. All the students successfully completed all the changes.

Each time the students ran the testing tools we captured the source code, timestamped it, and gathered testing statistics.

4. Data analysis

The experiment has just completed and we are in the preliminary stages of data analysis. Our initial analysis has 3 steps.. First we are examining methods for normalizing several confounding influences. Next we will determine whether the testbed exhibits signs of code decay. If it does, we will then look for relationships between our independent variables and the appropriate response variables.

4.1 Confounding Factors

As we discussed in Section 3.3, there are several factors affecting the experiment's internal validity. These include differences in programmer behavior, system complexity, and change request complexity. Currently we are examining strategies for normalizing these differences.

4.2 Measuring responses

While we are explore normalization strategies, we are also examining several response variables. These include programmer effort, number of test runs, effect of modifications, and various code complexity metrics.

4.3 Examining Potential Causes of Code Decay

We are currently examining the effect of the independent variables of the number of test runs needed to implement the change request. Although we haven't performed a complete analysis, we do see that neither the number of changes already performed, nor the number of developers who have already modified the system explains the number of test runs. There is some indication that when a developer modifies a system for the first time the modification is more difficult than later attempts to modify any implementation of the same system.

5. Summary

In this article we presented the goals of the code decay project. To help the project achieve its goals quickly, we have constructed a set of model code artifacts that we call the code decay testbed. Our approach was to develop several small systems, attempt to induce code decay in them, and use the resulting systems to evaluate ideas about code decay.

The testbed has several limitations and by itself may never generate definite conclusions. For example, several factors, such as differences in programmers, systems, and change requests, are not controlled. We are exploring approaches smooth these effects in our data. Also, because the artifacts are small they are unable to explain decay due to a system's module inter-relationships or process and organizational constraints. We will focus instead on decay due to internal structure and poor code comprehension.

Despite these problems the testbed has already proven useful to the project. In one case it directed a researcher developing code decay indices to consider the effect of code familiarity. That is, Developer A makes two changes to a system. If many other developers make changes in between, Developer A will be less familiar with the system when making the second change.

We have also noticed two different styles for implementing changes. In one case the developer thinks through the change identifies the necessary modifications, and makes them at one time. The other approach was to run the test tools, find N safety property violations and then make changes that reduce the number of violations. These styles reflect an approach of building quality in versus testing quality in. We are currently exploring the effect of these different styles on the modification histories of each implementation.

Finally, we are continuing to analyze the testbed, but have much more work left to do. Also, the data set is extremely small and we plan to conduct more replications.