A Case Study on Improving Quality During Legacy Software Maintenance Using a Heuristic

Michael J. Salé
Seidenberg School of CSIS, Pace University, Pleasantville, New York

***Abstract*—Legacy software is defined as software that contains no testing suite, is most-likely foreign to the developer, lacks meaningful documentation, and over time, has become difficult to maintain. Because of these flaws, developers frequently make modifications without any formal plan. This type of legacy software maintenance is problematic and often results in a reduction of software quality (e.g. more defects, code smells and rot, high levels of brittleness, lack of conceptual integrity, and bolt-on modifications).**

General guidelines for legacy software maintenance were developed in prior work, but none of these guidelines were evaluated during actual maintenance tasks. This paper contributes to the corpus of legacy software maintenance research by: presenting a novel, practical, legacy software maintenance heuristic; and evaluating the heuristic through a case study. The case study showed that an experimental group of developers using the heuristic improved software quality for a set of legacy software maintenance tasks compared to a control group of developers.

***Index Terms*—heuristic, legacy, software engineering, software maintenance, software quality**

I. Introduction

OVER the past fifteen years much has been researched and written on the topics of software development methodologies and frameworks such as Agile and Scrum. These frameworks focus on the development of new software projects, otherwise known as greenfield projects. In reality however, greenfield development is not the norm [1]. Almost all software development projects contain some amount of legacy code, and therefore, developers must integrate new changes into an existing system. These changes, referred to in this research as maintenance tasks, range in complexity from simple defect repair to more non-trivial development of new features. The problem that many developers encounter in this maintenance process, is that the code has been maintained by many developers over its lifespan, not all of who maintained the conceptual integrity of the system. The code often contains little to no formal automated testing suite and the associated development documentation has often gone without updates since the system's original design. As a result, developers often make changes to the system without any formal plan or guideline and do not have an understanding of how they will impact software quality. Furthermore, the longer that legacy systems continue to be maintained in this fashion, the more software quality suffers. Functional defects may be introduced, some of which may not surface for quite some time and bolt-on modifications usually introduce code smells and contribute to code rot, making the code much more difficult to maintain in the future. If developers were able to follow a heuristic when performing maintenance to legacy code, which maintained or increased software quality, the risk of contributing to software brittleness can be reduced. This heuristic's main features assist developers in understanding existing code, creating a testing suite for existing and new code, and measuring adequate test coverage for the maintenance task. This paper provides an overview and brief discussion of research performed for a dissertation on legacy software maintenance. The research was performed as a mid-sized case study in a college software engineering course. It sets out to demonstrate that using a heuristic to perform legacy maintenance tasks is not only practical, but improves overall system quality.

II. Research Problem and Questions

This research explores whether or not using a heuristic while performing maintenance tasks to legacy code improves the quality of the new code and maintains or improves the quality of the existing code. Organizations rely on information systems for the continued functionality of their business processes. These systems range from off-the-shelf commercial packages to custom software developed in-house. This research focuses on the latter. The roles that these systems play also vary, including simple process automation, customer relationship management, enterprise resource planning, and others. The lifetime of these systems may be as low as two years for small business supporting systems or as high as twenty-seven years for large administration systems, the average lifetime being about ten years [2]. Because of this long average lifespan, organizations must continually maintain these systems by adding new features, changing existing functionality, and repairing defects.

Often times, reengineering or starting from scratch is not an option for an organization, particularly when technical cost overhead must be kept to a minimum [1]. Management must be

This work was supported in part by a Stonehill College professional development research grant. Paper submitted for review on April 21, 2017.

M. J. Salé is an Assistant Professor of Management and the Management Information Systems Program Director at Stonehill College in Easton, MA. He is currently completing his DPD degree at Pace University (e-mail: msale@stonehill.edu).

able to justify reengineering by comparing the cost of replacement to the cost of maintenance [3].

A. Research Questions

This research evaluates these questions:

1. Does using a heuristic during legacy code modification improve the quality of the modifications?
2. Does using a heuristic during legacy code modification maintain or improve the quality of the existing system?
3. Is using the heuristic practical? Can a development team use the heuristic and perform maintenance in an similar amount of time as compared to a team not using the heuristic?

B. Research Hypotheses

1. The quality of the maintenance tasks performed by the experimental team using a heuristic will be higher than the quality of the maintenance tasks performed by the control team not using a heuristic.
2. The experimental team will be able to complete the same number of maintenance tasks using a heuristic as the control team not using a heuristic.

III. Prior Work

A. Defining Legacy

The term "legacy" was first used by Computer Scientist George Olivetti to describe code maintained by someone who did not initially develop the code. The important piece to the maintenance of this code is the difficulty in doing so. More modern definitions of legacy take this into account. Most recently, Michael Feathers, the author of "Working Effectively with Legacy Code" and a well-respected leader in software development training and leadership mentoring, defined legacy quite a different, but important way. Feathers' definition of legacy code is simply, "code without tests." His argument in favor of this definition is that code that lacks tests (automated or otherwise) is difficult to maintain. While lack of tests can certainly contribute to the difficulty of maintenance, there are several other factors which complicates the process of performing maintenance on legacy code [4].

The first factor is lack of documentation, which could take the form of user guides, in-line code comments, Internet forum posts (for open-source and commercial software), and other internal artifacts. Developers who are unfamiliar with a system can gain better understanding of its functions and structure by accessing this documentation. Another factor is a lack of the ability to communicate with either the original developers of the system or developers who have previously performed maintenance. Developers certainly have a community mentality and being able to share acquired knowledge with others to solve a problem is common. A wealth of understanding of a system can be gained by conversation and apprenticeship with other developers. It can help with both understanding how a component of a system functions or why the developer coded something the way he or she did.

This discussion leads us to a viable conclusion, software becomes difficult to maintain when you lack the conceptual integrity necessary to maintain it. Conceptual integrity can only be achieved when the structure of the system is simple to understand and consistent throughout. If developers performing maintenance on a system continue to lack the understanding of a system, it will most-certainly lead to code that follows structures unlike that of the existing system, bolt-on changes, and the introduction of code smells or anti-patterns.

Therefore, for the purposes of this case study, a developer is maintaining legacy code when:

- the code exists without automated tests (unit, characterization, or other),
- the system lacks external or in-line documentation,
- the developer is unable to communicate and learn from the original developer of the system or those who have previously maintained the system,
- the developer may lack sufficient domain knowledge of the system or a system with similar functionality

What does this mean for the traditional definition of legacy? It means that the age of the code base or the language in which it was written may have little to do with whether or not the system is legacy. Indeed, maintaining code that was written in older languages can be difficult, however, it is the characteristics listed above that will define legacy in this research.

B. Why Maintain Legacy Software?

Throughout this research many have asked, "Why would you want to continue to maintain legacy code?" There can be several reasons as to why an organization wishes to continue to maintain legacy code/systems, in their traditional sense. Many times the systems work satisfactorily and the organization sees no reason to change it. The costs of replacing the system are prohibitive or make little business sense. Retraining on the new system would be very costly. The current systems may meet regulatory standards. There are others. For whatever reason an organization continues to maintain legacy systems, the fact is that most developers will at one time or another have to perform maintenance to a system that they did not develop.

The speed at which the maintenance of legacy code is made tends to be much slower that an actual greenfield development process. Why is legacy code maintenance so much slower? Feathers states, "…How easy would it be to modify your code if it could bite back, if it could tell you when you made a mistake [4]?" In his jest, Feathers provides one of the cornerstones of this research. Legacy code maintenance is a difficult, time consuming, and risky process.

Understanding and maintaining legacy code has been a challenge to software developers for many years. Despite the advances in software reengineering practices, tools and IDEs, and guides to best software development practices, there are many cases when time and budget is too lean to apply heavyweight reengineering practices on a code base. However, while the time and money are lean, the risk associated with legacy code maintenance is not. If this is the case, there must be a way to minimize this risk. There have been many methods prescribed in literature as to how to effectively perform

modifications to legacy code. The aforementioned book by Feathers contains multiple guidelines and a search of the Internet will reveal many developers' opinions on how to conquer unknown code. However, until this point, no research has been performed on whether making modifications using a set of specific guidelines actually improves the quality of the modification and the existing legacy code.

C. Current Software Maintenance Practices

A 2001 case study on legacy maintenance in IEEE Software discusses the inherent issues with most software engineering practices in a legacy environment.

The first problem occurs in a development team's ***processes and practices***. On most teams, each developer's standards of quality, "doneness" of a maintenance task, and best practices vary widely. Furthermore, most developers do not have a focus on process improvement. Instead, developers have a goal of taking a change request, and completing it as quickly as possible so that they can move onto the next task. Lastly, most times, maintenance tasks are assigned to junior engineers, who oftentimes lack the experience with good engineering practices and can be fearful of changing existing codebases with which they have very little experience.

The second problem occurs in lack of ***testing***. Unit testing, test-driven development, and integration testing, not to mention automated testing suites, are generally not the strong suit of most software development groups. Many times test suites are bulky, lack sufficient coverage, and inconsistently maintained. Furthermore, many older legacy systems were developed before automated test suites became commonplace in the industry or before adequate test harness software was available for the language in which the system was written. This usually means there is no motivation to maintain or create a comprehensive test suite for the system. This results in maintenance tasks being implemented with no indication as to whether the change could have created defects elsewhere in the system. A lack of testing suite also seems to diminish a developer's understanding of existing code. In this case study, most of the developers used existing unit tests to aid in understanding what code did. Without unit tests and sufficient documentation, developers are left to their own devices in understanding code and the best way to make changes to the code.

The third problem is a result of the first two problems. This problem is ***software entropy and rot***. By having a lack of clear processes and practices for performing legacy maintenance tasks and little to no system or code testing strategy, software will begin a slow deterioration in quality where it becomes very difficult, if not impossible to maintain. A simple maintenance task becomes a major undertaking and possibly causes defects is multiple places in the system. This phenomenon has caused many organizations to have to upgrade to a new system costing them thousands or millions. This rot also furthers the difficulty that developers have when trying to understand how a system works. The accumulation of "band-aid" changes and maintenance tasks completed using "edit and pray" mentality erodes at the quality of the system to a point at which it can no longer serve its purpose.

D. Software Quality

This realization requires us to define what we mean by quality. As any developer who has worked on a team knows, there are many ways to implement code. This creativity is one of the perks of being a programmer. Unfortunately, this also means there are many opinions as to what makes "good" code. What one developer values in terms of code quality, another may completely dismiss. In fact, during the research, one of the developers felt that the best code was written compactly and concisely, while another developer thought that more verbose code was easier to maintain and read. To this end, there needs to be a functional definition of quality for this research. One of the major problems in defining software quality is that "everyone feels they understand it" (Crosby, Quality is Free, 1979) and it seems that not many achieve it. Historically, software quality has been defined, along with its metrics and attributes, by the ISO 9126-3 and the subsequent ISO 25000:2005 SQuaRE standards. By these standards, quality software features these characteristics: reliability, efficiency, security, and (adequate) size. Quality software has to satisfy both functional and non-functional requirements. To use a concise definition of quality from J.M. Juran's book, "Juran's Quality Control Handbook", "Quality consists of those products which meet the need of customers and thereby provide product satisfaction." and "Quality consists of freedom from deficiencies [5]." These two definitions pair up with the desire to meet both functional and non-functional requirements.

1) *Functional requirements*
   Functional requirements are rather easy to understand. A functional requirement is met when a system function accomplishes what is was set out to accomplish. A function is generally defined as a set of inputs, the behavior, and the outputs. Functional requirements are most-often tested with use cases. These functional requirements could be conveyed in a functional requirements document or by a user story in the agile process.
2) *Non-Functional requirements*
   Meeting non-functional requirements, however, has less to do with the specific functions or features of software, but more to do with specific criteria that are used to evaluate the operation of software. A popular model described by Ian Sommerville, divides non-functional requirements into three parts: product requirements, organizational requirements, and external requirements [6]. Somerville also explains that it is important for these non-functional requirements to be objectively tested. The three parts can be broken down into these subcategories:
   - *product requirements: usability, efficiency, and portability requirements*
     - e.g. The interface must be completely accessible via a mobile browser running in iOS or Android platform.
   - organizational requirements: delivery, implementation, and standards requirements
     - e.g. The new feature must be developed using test-driven development and

implemented during a scheduled maintenance window.

- external requirements: interoperability, ethical, and legislative requirements.
    - e.g. The new mobile application must not send any personal customer data to the vendor.

Sommerville's definition of non-functional requirements is just one of many. One source lists over fifty examples of non-functional requirements for software.

E. Defining Quality for the Case Study

For the purposes of this case study, the researcher rated quality software as software that meets both functional and non-functional requirements in the following ways:

- The customer or product owner has accepted the user story and the functional requirement has been met.
- The modification was made with adequate testing. Adequate testing is characterized as tests that show ample coverage (at the code, feature, and use-case levels). The three types of testing include characterization tests, unit tests, and acceptance tests [7].
- The modification was made to maximize structural design simplicity and maintain or improve conceptual integrity of the code.
- The modification was made to maintain or improve understandability.

In order to enforce functional and non-functional maintenance quality, a guideline, which we describe as a heuristic was given to developers to use for each maintenance task.

IV. The Heuristic

The heuristic used by the developers is comprised of five phases:

- Learning phase: getting a grasp on what the code does, and what areas of the code will be affected by changes
- Assessment phase: assess the potential risk in making the proposed changes,
- Testing phase I: determine test coverage needs and write tests for current and proposed functionality,
- Coding phase: write just enough code to implement the change,
- Testing phase II: exercise the affected areas of the system and the new functionality (if applicable) with acceptance tests (automated or manual).

Each phase of the heuristic suggests several methods developers can use. Due to the fact that each development task and system is unique, not all methods will or should be used. It is left to the discretion of the developer to select appropriate methods in each phase. One caveat of heuristics is that using a heuristic does require some level of expertise on the user's part. Some of the methods may be foreign to the developer; but with repetition and guidance from other developers (using agile techniques such as pair programming), methods become more familiar and developer use them more confidently. A discussion of each phase follows. Not all methods for each phase are discussed. Only methods used by the developers are included. More detailed descriptions of each method can be found at: https://studentsstonehill.sharepoint.com/portals/hub/personal/msale/A-Case-Study-on-Improving-Quality-During-Legacy-Software-Maintenance-Using-a-Heuristic.

A. Learning Phase

The goal of the learning phase is for developers to gain as much understanding of the system, the features on which they will be performing maintenance, and the structure of the code. The learning phase helps developers understand the system at both a functional and code level. At first, this phase could take quite a long time to complete due to the fact that developers may be seeing and/or using the system for the first time. As time goes on, developers will build a cumulative understanding of the system, its structure, and how major classes and methods work and it will just be a matter of learning uncharted sections of the code. The techniques for this phase are:

- Setup Environment and Build Project – Helps the developer understand the development environment for the project by having them setup a full development environment of the system, make a trivial change, and build and deploy the system to a test server.
- Use and Research of the System – Asks the developer to use the feature(s) in the system that requires maintenance. This includes understand the user interface and its nuances, inputs, perceived processing, and outputs.
- Code Review – Code review asks developers to collaboratively review the code that requires maintenance. A lightweight code review process was prescribed to developers. In teams of two or three, developers reviewed the code which would ultimately be changed. Modeled after the agile practice of pair programming.
- Class and Sequence Diagrams & Class-Responsibility-Collaborator Cards – These two techniques suggested that developers manually create or use the IDE to create class or sequence diagrams and possibly construct CRC cards for the affected classes. While CRC cards are usually constructed before classes are written, this exercise could give developers a better visualization of how classes collaborate and what their responsibilities are.
- Documenting and Commenting – Pivotal to this phase is the documentation of the knowledge that the developers gain through these techniques. As developers learn about the system they should document their findings either externally in a journal or wiki, or place relevant inline comments in the code.

B. Assessment Phase

Once the developer has an understanding of the structure and functions analyzed in the learning phase, an assessment of risk of the maintenance task can begin. The risk is evaluated at two

levels: the risk of the feature or code that you may be adding, and the risk level of the affected areas of the current code. The end of this phase is marked by the developer assigning a risk level to the maintenance task. Each maintenance level will require a different level of testing in the next phase. The techniques for this phase are:

- Code Effect Analysis – This technique helps developers determine how far-reaching the effect may be of their maintenance task. It does this through analyzing return values, identifying inheritance, and finding global variables. Effect levels are separated into first, second, and third-level effects, each carrying a higher risk weight. The more far-reaching an effect, the more tests required.
- Code Smells – Code smells can indicate trouble ahead or simply just suggest the possibility of later refactoring. In this heuristic, code smells are just one of the many warning signs that may indicate code brittleness. Approximately 25 code smells can be detected either manually using a code smell worksheet or automated IDE tools.
- Assigning a Risk Level – The ultimate goal of this phase is to determine the perceived risk of the maintenance task. A developer is going to assign a higher risk level to a method in a superclass with code smells as compared to a well-written standalone method. Risk levels assigned as low, moderate, or high.

C. Testing Phase I

Once the riskiness of the maintenance task is established, a developer has a better idea of what level of caution is required for the change. Prior to creating any tests, a testing harness must be created in the project if it does not already exist. This is outside the scope of the heuristic but is necessary to create, run, and store tests. Based on the risk level, a developer will create the necessary tests. The main goal of this phase to create what are called characterization tests. Characterization tests are similar to traditional unit tests that should be written during the initial development of a system, but characterization tests ask developers not to test the often unknown intended behavior of affected code, but instead to test the actual behavior of existing code. Once an ample set of characterization tests are created, they can be run after making code changes to ensure that change did not change the behavior in any unintended way.

Difficult to test methods may require refactoring or a developer may use what is called vice or logging seam testing. This testing technique allows a developer to create tests based on the contents of a log file produced by inserting logging statements inside the classes. Once the class is executed, the results of the inserted logging statement are compared against the asserts of the test. This is very useful when a major refactoring of a difficult to test method is not possible.

For each method that is tested, a developer wants to create enough characterization tests so that all basic functionality of the method is covered and the behavior of the method is understood. A good rule of thumb is to write enough tests to cover specific inputs and outputs that are known to occur while paying attention to edge cases if possible. To determine test coverage, an automated coverage tool in the IDE can be used.

D. Coding Phase

Once the appropriate testing has been created, the developer can then begin writing the production code. As stated in the definition of non-functional requirements, one of the main goals of maintaining legacy code is to ensure that the code continues to be maintainable. One of the most important practices a developer can adopt to create maintainable code, is to write "clean code." There is no one definition of what clean code is and every developer has a personal opinion about what it means to write clean code. However, the development community can reasonably agree on one statement. "Clean code is code that is easy to understand and easy to change." Michael Feathers summarizes clean code as, "Clean code is code that is written by someone who cares. [4]" There are certainly some techniques that can help a developer to write understandable and maintainable code. The good news is that most developers that are learning how to write code in most computer science programs are being taught to write following many of these practices. The following are the techniques in the heuristic:

- Test-Driven-Development – Introduced in 1999, test-driven-development is a development process whereby unit tests are written for a new piece of production code before the actual code is written. The result of adopting test-driven-development is code that accomplishes just enough to meet the requirements of the feature being added or changed. The entire process is based on a repetition of a very short development cycle where requirements are turned into specific test cases and then enough code is written to make those test cases pass. This ensure that no code is added to the system unless it meets some requirement [8].
- Single Purpose and Non-Redundant Code – Clean code should be focused and should conform to the Single Responsibility Principle (SRP). Each module or class of the code should have responsibility over only one aspect of functionality provided by the software. Code should also be written so that a change to any single element of the system does not require a change in other logically unrelated elements. Developers should follow the Don't Repeat Yourself (DRY) principle when writing new code.
- Natural and Pleasing Code – Code should be written purposefully to accomplish a task or solve a problem and should not look awkwardly written. If a developer has to use a workaround to accomplish something in a program, chances are insufficient thought has been given on how best to use code so that it solves the problem in a simple, straightforward way. Code should also be pleasing to read and should be written as simply as possible. Developers should focus on writing the simplest code that make the software work and accomplishes what has been set out.
- Code with Minimal Dependencies – The more

dependencies code has, the harder it will become to maintain in the future. Writing code with the smallest amount of dependencies not only makes the code easier to maintain, but it is also courteous to future developers that may have to maintain the code.

E. Testing Phase II

Once production code is written and submitted, it must then be tested again in a different manner. This phase consists of functional tests and user acceptance tests. These two test types are often confused. Here are the definitions of these tests:

- Functional Tests – Quality Assurance uses functional tests to ensure that the new or changed functionality of the system works.
- User Acceptance Tests – A user acceptance tests are used by product owners or customers to verify if specific requirements work for them. This test type seems very similar to functional tests, but there are situations where a feature technically works, but it is not acceptable from the user's point of view.

V. Research Environment/Methodology

This research primarily took place over a four-month academic semester. The research setting was an undergraduate computer science software engineering capstone class consisting of eight senior class students, six of which assumed the role of software developer, one who acted as a scrum master, and one who assisted with user acceptance testing and writing test cases. The researcher assumed the role of the development team's product owner.

The six developers were divided into two groups. The groups were not formed randomly but so that each group had equal skill. The experimental group (Team A) would utilize the legacy modification heuristic for all maintenance tasks while the control group (Team B) was allowed to make modifications without much guidance.

Maintenance tasks were performed on software called JSPWiki, an open-source WikiWiki engine, built around standard JEE components and maintained as a Top Level Project by the Apache Software Foundation [9]. The project was managed using a mix of Agile and Scrum principles [10]. The developers were issued a series of maintenance tasks in the form of user stories.

The project was guided by the researcher and a faculty member in the Computer Science department. The development portion of the semester was separated into six sprints or development cycles, where at the end of each sprint, a retrospective was held and submitted code was reviewed by the researcher.

A. Data Sources for the Case Study

1) Retrospectives and Stand-Up Meetings

The two primary sources of data used for this case study were the end of sprint retrospectives and the stand-up meetings which occurred three times per week. Sprint retrospectives are an agile practice usually performed at the end of a development sprint. The goal of a retrospective is to provide a time where teams can reflect on what went well, what did not go well, and what is still unknown at the end of each sprint. It is designed to help teams identify what to start doing, continue doing, or stop doing. In this case study the sprint retrospectives were not only used for these purposes, but as an opportunity for the researcher to gather data and ask important questions. Sprint retrospectives were held for each team separately to prevent contamination. Retrospectives also provided time for the teams to present their progress and demonstrate completed user stories to stakeholders. All retrospectives were recorded.

Stand-up meetings, also called the daily scrum, are another agile practice whereby a product owner or scrum master holds a brief team meeting to facilitate communication between the entire team. Stand-up meetings are not to be used as troubleshooting or demonstration sessions, but to maintain team progress. At each meeting, the team members answer three questions.

1. What did you do yesterday?
2. What will you do today?
3. Is there anything stopping you from getting your work done?

The meeting is not designed to merely provide status, but to also bring to the attention of the scrum master or product owner, any issues that need to be dealt with in order to maintain productivity.

2) Journal Entries

Throughout the case study, developers were asked to journal their progress and any important conversations had within the team during programming sessions. Journaling was required both for the team members to record important facts for later reference and for the researcher to have further explanation for decisions made by teams. A journaling frequency was not enforced, but all journaling had to be done in one central location – a course learning management system. Journal entries were reviewed after every sprint post-retrospective. Journaling became increasingly important for Team A as a documentation tool during the Learning Phase of the heuristic.

3) Code Review

Code review was another source of data for the case study. The researcher reviewed code checked into the revision control system (SVN) at the end of each sprint to review how each team had performed their changes. There was also a more formal code review halfway through the semester whereby the researcher and research advisor reviewed some of the code written by the teams during the fourth sprint. Code review was helpful in tracking team progress and helped the researcher understand more fully, the direction the teams were heading during each sprint.

VI. Sprint and User Story Summaries

Table I displays how the user stories were assigned throughout the six sprints, a short description of each user story, and each team's "doneness" level at the end of each sprint for each user story. Many of the stories not completed within one sprint are often moved to a subsequent sprint for completion. Observations and discussion of results will follow. Most

TABLE I
USER STORY SUMMARIES

Sprint/ Story #	User Story	Story Status at End of Sprint	
		Team A	Team B
1/1	As a privileged user I would like to CRUD user accounts.	*Doing*	*Doing*
1/2	As an administrator I would like to be able to set the entire wiki as "closed" so that new users cannot self-register.	*Doing*	*To Do*
1/3	As an administrator I would like to create, retrieve, update, and delete (CRUD) "n" user groups.	*DDD*	*DDD*
1/4	As a privileged user I would like to control user group permissions from the administrative interface.	*To Do*	*To Do*
2/1	As a user I would like to be able to save a post as a draft and send it when I am ready.	*Doing*	*Doing*
2/2	As a user I would like to be able to save a post as a template for future use.	*To Do*	*Doing*
2/3	As a user I would like unsent posts to auto-save as a draft every 5 minutes.	*To Do*	*To Do*
2/4	As a privileged user I would like to control user group permissions from the administrative interface.	*To Do*	*To Do*
2/5	As a privileged user I would like to CRUD user accounts.	*Doing*	*Doing*
3/1	As a user I would like to be able to save a post as a draft and send it when I am ready.	*Doing*	*Doing*
3/2	As a user I would like to be able to save a post as a template for future use.	*Doing*	*Doing*
3/3	As a user I would like unsent posts to auto-save as a draft every 5 minutes.	*To Do*	*To Do*
3/4	As a privileged user I would like to CRUD user accounts.	*DD*	*To Do*
4/1	As a user I would like to be able to save a post as a draft and send it when I am ready.	*Doing*	*Doing*
4/2	As a user my posts should not be re-versioned if only minor edits are made by the same user.	*Doing*	*To Do*
4/3	As a user I would like to be able to mark my posts as "unsearchable."	*To Do*	*To Do*
4/4	As a privileged user I would like to be able to rename an attachment.	*To Do*	*To Do*
4/5	As a user I would like unsent posts to auto-save as a draft every 5 minutes.	*To Do*	*To Do*
5/1	As a user I would like to be able to save a post as a draft and send it when I am ready.	*Done*	*Done*
5/2	As a user my posts should not be re-versioned if only minor edits are made by the same user.	*Doing*	*To Do*
5/3	As a user I would like to be able to mark my posts as "unsearchable."	*To Do*	*To Do*
5/4	As a user I would like unsent posts to auto-save as a draft every 5 minutes.	*To Do*	*To Do*
6/1	As a user I would like to be able to save a post as a draft and send it when I am ready.	*DDD*	*DDD*
6/2	As a user I would like all of my drafts to be versioned.	*To Do*	*DDD*
6/3	As a privileged user I would like to CRUD user accounts.	*DDD*	*DDD*
6/4	As a user my posts should not be re-versioned if only minor edits are made by the same user.	*DD*	*DD*
6/5	As a user I would like unsent posts to auto-save as a draft every 5 minutes.	*DDD*	*DDD*

Done = user story is completed but has not been tested by QA nor accepted by product owner, DD = Done, Done = user story has been accepted by product owner but not QA, DDD = Done, Done, Done = user story has been accepted by product owner and has successfully passed QA testing.

observations were gathered during stand-up meetings and sprint retrospectives.

VII. OBSERVATIONS AND RESULTS

A. Observations and Analysis

The following are a small sample of observations taken from the full case study. Remember that Team A is the experimental team using the heuristic and Team B is the control team not using the heuristic.

1) Code Learning

At the beginning of the case study, both teams wanted to know what the current code did and how it was structured. Team A was able to use some of the methods to begin a structured review of the code that was being changed as well as code that may be affected by the change. Team B took a less-structured approach to learning the code base and only paid attention the code being changed. Team A later stated that the code learning methods was one of the most important parts of the heuristic and increased their confidence in the stability of their maintenance tasks [11].

2) Practicality

Throughout the case study, the researcher worked with Team A to make the heuristic as practical as possible. The initial heuristic was a long document in prose form. In order to effectively and continuously use the heuristic, the researcher suggested and constructed a checklist to use during maintenance tasks. Checklists have been found to be an effective tool for developers performing maintenance tasks on code [12].

3) Agile methods

Some of the methods in the heuristic required Team A members to utilize certain agile methods such as pair-programming and test-driven development. The developers found these experiences to be very valuable to their development process. For example, there were numerous times when team members used unit testing as a tool to learn what a piece of code did. Other times, team members used pair-"code review," a take-off on pair-programming, which allowed two or more developers to work together over each other's shoulder to review existing code and document it's functionality. These practices showed to be beneficial for the team and the overall understanding of the code.

4) Bolt-on modification

The true benefits of the heuristic came out when the teams were assigned user stories which added new features to existing functionality. Team A used the heuristic methods such as writing CLEAN code, creating unit tests for already existing code to learn about its functionality, and assessing risk of the change throughout the system (analyzing dependencies, etc.). This resulted in changes that were well-integrated and maintained the structural integrity of the existing system. A developer making changes at a later time would not be able to see the "seams" where the new code was inserted. A new developer would have a comprehensive suite of unit tests built around the new functionality and the methods affected by the maintenance task, making his job much easier. While the maintenance tasks were also completed by Team B, they were made using a technique of "risk avoidance." One of the developers stated during a

retrospective that the team did not touch code they did not understand and "just got in there, did what needed to be done, and got out." This seems like a haphazard methodology, but is very common for developers working in legacy systems [11], [4].

5) Comparison of Team Productivity

One of the questions in this research was whether using the heuristic was practical or caused tasks to take more time. As you can see in Table I, the teams made similar progress by the end of each sprint, despite the fact that Team A was required to use many methods in the heuristic, some taking extensive time. Each week, both teams spent similar cumulative time writing code according to the burn-down charts maintained by the scrum master.

6) Code Quality

The two major questions in this research were:

1. Does using a heuristic during legacy code modification improve the quality of the modifications?
2. Does using a heuristic during legacy code modification maintain or improve the quality of the existing system?

From the sprint retrospectives, analyzing of code commits by the researcher and the research advisor, and from QA results, there is a clear difference in quality between the Team A and Team B code bases. There were two major instances in which Team B used a bolt-on approach to adding new functionality to the system. See stories 3/1 and 6/2 in Table I. The new functionality was not integrated in a way that followed or maintained the structural integrity of the system. These two particular modifications caused there to be duplication of code in multiple places in the system. By definition, this is a code smell, which increase code entropy. A developer modifying the same set of methods in the future would find himself having to make changes in multiple places to accomplish one task. Lastly, because Team B did not use unit testing, they were also unsure as to whether the bolt-ons they added affected other areas of the system. A quote from one of the developers was, "That's QA's job."

VIII. Conclusion

The analysis of the results provides support for the research hypotheses and affirms the research questions.

1. Does using a heuristic during legacy code modification improve the quality of the modifications?
   Yes, the use of a heuristic during legacy code modification by the experimental group improved the quality of those modification using the definition of quality discussed in Part III, Sections D and E. These modification were successfully integrated into the system without causing code entropy.
2. Does using a heuristic during legacy code modification maintain or improve the quality of the existing system?
   Yes, through the practice of writing unit tests for existing code before making any further changes, as well as code refactoring practices, increased the quality of the existing system through reducing code smells, and incrementally building a comprehensive automated test suite.
3. Is using the heuristic practical? Can a development team use the heuristic and perform maintenance in an similar amount of time as compared to a team not using the heuristic?
   Yes, the experimental team was able to be, on average, as productive as the control team working a similar number of hours per week.

References

[1] V. Rajlich, "Comprehension and Evolution of Legacy Software," in *Software Engineering, 1997., Proceedings of the 1997 (19th) International Conference on*, 1997, pp. 669–670.

[2] T. Tamai and Y. Torimitsu, "Software lifetime and its evolution process over generations," *Proc. Conf. Softw. Maint. 1992*, pp. 63–69, 1992.

[3] H. M. Sneed, "Planning the Reengineering of Legacy Systems," *IEEE Softw.*, 1995.

[4] M. Feathers, *Working effectively with legacy code*. 2004.

[5] J. A. Defeo, *Juran's Quality Handbook: The Complete Guide to Performance Excellence, Seventh Edition*. McGraw-Hill Education, 2016.

[6] I. Sommerville, *Software Engineering: (Update) (8th Edition) (International Computer Science)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[7] M. Feathers, "Characterization Testing." [Online]. Available: https://michaelfeathers.silvrback.com/characterization-testing. [Accessed: 27-Oct-2016].

[8] K. Pugh, *Lean-Agile Acceptance Test-Driven Development*. Net Objectives, 2010.

[9] "Apache JSPWiki." [Online]. Available: http://jspwiki.apache.org/. [Accessed: 01-Jan-2015].

[10] K. Beck and C. Adams, *Extreme Programming Explained*. 2004.

[11] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How Do Software Engineers Understand Code Changes?: An Exploratory Study in Industry," *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, pp. 51:1–51:11, 2012.

[12] M. Keil, L. Li, L. Mathiassen, and G. Zheng, "The influence of checklists and roles on software practitioner risk perception and decision-making," *J. Syst. Softw.*, 2008.