Lehman’s Laws in Agile and Non-agile Projects

Kelley Duran, Gabbie Burns, Paul Snell
Rochester Institute of Technology
United States of America
kad8166@rit.edu, gbb5658@rit.edu, pxs2090@rit.edu

Abstract—Software team leaders and managers must make decisions on what type of process model they will use for their projects. Recent work suggests the use of agile processes since they promote shorter development cycles, better collaboration, and process flexibility. Due to the many benefits of agile processes, many software organizations have shifted to using more agile process methodologies. However, there is limited research that studies how agile processes affects the evolution of a software system over time.

In this paper, we perform an empirical study to better understand the effects of using agile processes. We compare two open source projects, one of which uses a tailored agile process (i.e., Xtreme Programming) and another that has no formal process methodology. In particular, we compare the two projects within the context of Lehman’s Laws for continuing growth, deciding change, increasing complexity, and conservation of familiarity. Our findings show that all four of the laws hold true for the project that uses an agile process and that there are noticeable differences in the evolution of the two projects, many of which can be traced back to specific practices used by the agile team.

I. INTRODUCTION

From the time that software is initially released it begins to decay, even with developer efforts to prevent the effects of aging. Research in the area of software evolution has explored software aging and proposed techniques to minimize or delay its impact [11], [19]. Some of the work in the area has suggested that software process elements have an impact on the evolution of systems [9], [16]. Agile processes are of particular interest in this respect, but have not been fully investigated by researchers.

The agile development process promotes shorter development cycles, teamwork, collaboration, and process flexibility to manage risks and customer needs while developing a software product. These characteristics distinguish the agile process from other traditional development processes like waterfall. Since 2001, when a group of programmers wrote the Agile Manifesto to define agile development, proponents of agile development claim that software developed using agile processes can better withstand the pressures that threaten the quality of the software. Though much of the work today focuses on the benefits of agile during the initial development phases, recent studies have started to investigate the effects of agile processes on software evolution [10].

Within the development community, there is much speculation about whether or not agile processes are “better” than traditional process models. Previous research [10], [16] has suggested that software process elements have an impact on evolution. Based on that idea, we believe that the differences between agile and non-agile process models could have a significant impact on the evolution of otherwise similar projects. Sindhgatta et al. [21] were some of the first to study an agile project through the lens of Lehman’s laws on software evolution. In that work, they found that most of the laws proved valid for the agile project, but to the best of our knowledge there have been no studies directly comparing the evolution of two projects using different processes [21]. Even if we can use Lehman’s laws to describe agile as well as non-agile projects, they could still systematically vary in terms of quality and rate of decay. To help developers make educated decisions when selecting a process methodology, we need empirical evidence that can attest to the direct tradeoffs between them.

To address this, we performed a case study comparing the evolution of two open-source projects from a scientific domain. We picked two projects that were similar in all ways except for the process methodology used. For an example of agile development, we used the Cancer, Heart and Soft Tissue Environment (Chaste), which has been cited as an example of using agile processes in the field of computational biology [20]. As a comparison project, we used the Biochemical Algorithm Library (BALL). Both projects are written in the same language and fall within the same domain, but the developers of BALL do not use a structured process model [1], [2], [15]. We believe that comparing the two projects will give us more applicable insight into how the agile processes affect the evolution of the code. By using Lehman’s laws as the basis of our measurement, we give our findings a common ground for comparison with other work as well. Our work contributes to the greater body of software engineering knowledge by improving our understanding of the long-term effects of agile processes and giving more basis for comparing and contrasting agile and non-agile methodologies.

The rest of the paper is organized as follows: Section 2 provides background on previous work related to evolution of agile and non-agile projects. Section 3 discusses the research questions, and the rationale for metrics being used to answer each research question. Section 4 briefly discusses how we collected the data, and describes both open source projects in more detail. Section 5 discusses the results and findings we had for each of the metrics we chose. Section 6 discusses threats to validity and Section 7 concludes the paper.
II. RELATED WORK

In the 1970s Belady and Lehman [8] studied the os/360, an IBM operating system for a span of 13 years with over 20 releases, and formulated 8 laws of software evolution. Belady and Lehman at that time, suggested that there was a lot to learn in the science of program creation and maintenance. Their work made it clear that studying the evolution of software systems would allow us to perfect the science of developing software, and to understand what maintenance activities could mitigate the effects of software evolution.

Since the publication of the Belady and Lehman paper in 1976, numerous studies have been done observing the laws of software evolution in a variety of contexts. Godfrey et al. [12] studied the evolution of software in the open source context by looking at the evolution of the Linux kernel. They found that the Linux kernel has been growing at a super-linear rate for several years instead of growing at slower rates as the Linux code base increased. However, by looking closely at various sub systems, they found several subsystems including drivers were one of the factors that caused the growth to be super-linear, and the drivers subsystem grew the fastest and comprised for 60% of the entire Linux system. Godfrey et al. [12] also observed that Lehman’s third law of self-regulation, which states that the incremental effort spent on each release remains constant throughout a systems lifetime [18], was not being followed as there were periods where there was significantly more effort. They proposed that this change in effort was a consequence of using the open source process model. Godfrey’s et al. [12] study shows that the process model being used for development may impact evolutionary patterns of a project.

Capiluppi et al. [10] were the first to perform a measurement-based study on the evolution of software created using an agile process, looking at the size and complexity of the source code over time. In their case study, they took samples of a codebase four times per month to collect source code, unit tests, and code check-in information over the span of 2.5 years. To give context to the quantitative data, they also looked at the team’s records of progress and productivity, retrospective notes, and did an observational study. They found the growth was positive during most periods, but there were brief periods of less growth. The growth rate was higher for the lines of code than the number of files or directories. The differences between the in growth was influenced by reorganization and a reduction in staffing, rather than constraints from increasing complexity. The team members still produced code at the same rate, but with fewer developers. Results also showed the system analyzed had far fewer complex items compared to other studies looking at code that was developed under a non-agile process. The reduced complexity comes from the team’s commitment to refactoring. The study does not specifically discuss Lehman’s laws, but they do show increasing complexity and continuous growth. Our work studies those and others of Lehman’s laws specifically and compares two projects, rather than study only one.

A few years later, Sindhgatta et al. [21] were the first to study evolution using Lehman’s laws to understand the evolution of software development with a team using Scrum as an agile process. They divided the laws into four different groups: growth and complexity, self-regulation and conservation, increasing complexity, and declining quality. The team developed during 25 iterations over 15 months, ending with the first major release of the product. This study found all of Lehman’s laws held true. We study only a subset of Lehman’s laws, but we select two similar projects and compare them directly, rather than compare against the results of previous studies.

Singh and Singh [22] studied two non-agile open source projects and the applicability of all eight of Lehman’s laws. They found that the laws related to increasing complexity and continuous growth were validated by the metrics. One of the projects (HSQldb) had 17 versions and the other (HoDoKu) had 11. They use CBO and LCOM to measure complexity, as these two metrics can be used to measure the complexity of object-oriented systems. Their work shows that open source projects behave in similar ways as non-open source projects and that the Lehman’s laws apply. This paper is most similar to our own work, though our focus is agile projects not just open source and we make a comparison between different types of projects.

Though our work differs from these previous studies in several key ways, several of our metrics were inspired by the metrics used in these papers. We see our own research as a logical extension in the field, taking the current ideas in a new direction.

III. OUR APPROACH

To investigate the relationship between agile processes and evolution, we ask the following research questions:

• RQ1: Do Lehman’s Laws for growth, change, complexity, and familiarity hold true for an agile project?
• RQ2: Is there a significant difference in the way that agile and non-agile projects evolve?
  – RQ2.1: After controlling for growth, does the agile project show a higher rate of change than the non-agile project?
  – RQ2.2: Does the complexity of the agile project increase at a slower rate than the complexity of the non-agile project?
  – RQ2.3: Does the agile project show a more uniform level of developer familiarity than the non-agile project?

We narrowed our scope to focus on a subset of the laws that we thought would be particularly relevant based on properties of agile development. Below we discuss the reasoning for each law studied and explain the metrics used to assess them.

A. Growth and Change

We followed the lead of Sindhgatta et al. [21] and combined Lehman’s first and sixth laws for the purpose of analysis.
These two laws (Continuing Change and Continuing Growth) are directly connected, as growth in the system is a direct result of change. The law of Continuing Change describes the fact that a software system must continually adapt to its environment in order to meet user needs [17]. The law of Continuing Growth states that in particular, a system must continue to add new functionality over time [8]. Practically, these laws are often measured in terms of the amount of churn and size.

Agile processes such as eXtreme Programming (XP) incorporate refactoring as a step in the development process. The idea is that making smaller, frequent corrections to improve the design of a component is better than large refactoring efforts at longer intervals. In XP’s test-driven development cycle, refactoring occurs for every feature after the initial implementation [20]. We would therefore expect an agile project would exhibit a higher degree of change for each feature: one set of changes for implementation and one for refactoring. Because the second set of changes is for rework, we would also expect the higher degree of changes does not mean a higher degree of growth. The frequent refactoring may even reduce the amount of growth because it regularly trims out duplicate code and other design issues.

To assess system growth, we looked at the size at various intervals and then calculated at the rate of change over time. We measure size with the following metrics:

- M1: Number of LOC (per release) [10], [22]
- M2: Number of files (per release) [10]

Two different metrics were used initially to capture different aspects of the system’s growth and study it from multiple perspectives. However, we determined that the two metrics are strongly correlated at 0.97 for both projects, and therefore they can be used interchangeably with the same meaning. To assess system change, we used:

- M3: Code churn (per release)

The churn metric measures the number of lines of source code added or deleted based on the repository history. The churn metric is very similar to one used in [21], where they look at LOC modified and the number of files modified. In current version control tools, changing a line appears in the history as an addition and deletion which means the final number will include many double-counted lines. However, the double-counted lines is a consistent factor for all of the releases and projects measured, so we do not consider it a problem for using the churn metric.

B. Increasing Complexity

Lehman’s second law is the law of Increasing Complexity, which states that as a system grows, the level of complexity will continue to increase, unless there is a deliberate effort to reduce it [17]. As discussed above, agile projects are refactored continually throughout the development process. Refactoring reduces the amount of code complexity through simplification and cleaner design. In this instance, we would expect the complexity to increase at a slower rate under agile development because there are regular efforts to reduce it. In a non-agile project on the other hand, the complexity would steadily increase as predicted by Lehman with the possibility for sudden dips, which should correspond to a major refactoring or similar effort.

We follow the approach of Singh and Singh [22] in measuring complexity and used the following metrics to measure our projects’ complexity:

- M4: Coupling Between Objects (CBO) [22]
- M5: Lack of Cohesion Methods (LCOM) [22]

Each of these metrics is captured on a per-class basis. The CBO of a class counts the number of other classes with which it is coupled. LCOM is one of several possible cohesion metrics and counts “the set of methods in a class that are disjoint with respect [to] members of a class being accessed by them [22].” After collecting the data for our projects, we tested for correlation and found there was not a strong relationship between these and any of the size metrics, though there a strong correlation between CBO and LCOM (see Table I).

<table>
<thead>
<tr>
<th>Metrics Chaste Ball</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of files with SLOC</td>
</tr>
<tr>
<td>CBO with LCOM</td>
</tr>
<tr>
<td>number of files with CBO</td>
</tr>
<tr>
<td>number of files with LCOM</td>
</tr>
<tr>
<td>SLOC with CBO</td>
</tr>
<tr>
<td>SLOC with LCOM</td>
</tr>
</tbody>
</table>

C. Conservation of Familiarity

Lehman’s fifth law (Conservation of Familiarity) is more complicated to trace, but particularly relevant to agile processes. This law has several parts. The first part states developers must maintain a certain level of familiarity with the system in order to effectively work on it. If the system grows too big or too quickly, then they will not be able to and so the second part states the average growth of the system over time remains constant [17].

We were inspired to study this law after reading the work of Sindhgatta et al. [21] who frame it in relation to the phenomenon of code ownership. Agile projects place a strong emphasis on collective code ownership, so any member of the team can feel comfortable working on any part of the system [20]. When examining the applicability of this law, we focused on the level of familiarity of the system for all of the developers. With an agile project, we would expect to see a more even distribution of familiarity within the team and a higher average level of familiarity than that of developers on a non-agile project.

Measuring the familiarity for the system was more challenging and involved experiments with several different metrics. Finally we settled on the following metric:

- M6: Unique knowledge

The unique knowledge metric came to us from a particular tool called “Git by a Bus” [5] and determines the extent to which individual developers are the only ones working on parts
of the system. The metric calculates the total lines of source
code that were written by an individual and were then never
touched by others on the team. Using this metric, we would
expect good familiarity to mean lower uniqueness scores for
all of the developers.

IV. DATA COLLECTION

This section addresses two areas: the projects being ana-
lyzed and the paper's methodology. The methodology section
will explain and illustrate our process of gathering all the
metrics we needed to answer our research questions.

A. The Projects

Two open-source, academic-based projects were selected
as the basis for this study. Both projects are bioinformatics
libraries written primarily in C++. The first project, Chaste
(Cancer, Heart, and Soft Tissue Environment), was selected
because of the team's published work on their use of a
modified eXtreme Programming (XP) approach [20]. The
second project is BALL (Biochemical Algorithms Library) and
was selected based on its similarities to Chaste. In BALL's
documentation there is no mention of process activities, so
we contacted the development team, who told us they never
implemented any formal development processes. Any process,
particularly an agile one, requires discipline and knowledge of
software engineering concepts from the developers. We find
this is often not the case on computational science projects,
which are frequently developed by domain experts and use a
more ad-hoc process [13]. Both projects are also developed
for research and have a lower user base. Following are brief
descriptions of each project.

1) Chaste: Chaste [3], [4], [6] is an open-source exten-
sionable library that focuses on computational physiology and
biology. This library contains the ability to create cell and
tissue models. Chaste( agile) is developed by the Computa-
tional Biology Group at the Department of Computer Science,
University of Oxford. This project has over 10,448 commits
made by 48 contributors [3], [4], [6]. There are 287,613 lines
of code in the most recent version and over eighty percent
of that code is written in C++. Their repository is actively
maintained, with 81 commits in March 2013. There have
been 8 software releases over the last four years. Chaste’s
agile methodologies include pair programming, iterative bursts
with frequent planning and retrospective meetings, tests being
created for every piece of code written [3].

2) BALL: BALL [1], [2] is an open-source extensionable
library in the domain of Computational Molecular Biology and
Molecular Modeling. BALL(non-agile) provides data struc-
tures necessary for Molecular Mechanics, analyzing protein
structures, and visualization. This library is developed by
three groups located at Saarland University, University of
Tübingen, and University of Mainz. This project has had
18,184 commits across 27 branches. There are 249,180 lines
of code. Over ninety five percent of the code is written in
C++. The repository has had 10 releases over 11 years, with
the most recent release on January 28, 2013. The length of
time between releases for BALL(non-agile) is less regular and
ranges from 28 days to 773 days.

B. Methodology

To extract the data we needed for this project, we had to
employ several different tools. See Figure 1 for a high level
overview of our process. In the figure, each step is numbered
1-4 which are described in more detail below.

1) Step 1: We went to the websites of each project and
downloaded each release version. Each release contained
source code folders, a README.txt, and various library files.

2) Step 2: We went to each project’s Git repositories and
accessed their version control histories.

3) Step 3: In this step, we had to use several different tools
to extract different metrics. All of the gathered metrics were
put into an excel sheet for further analysis to be done in step
4.

The Understand tool [7] was used to create metric databases
for all releases in both projects. The Understand databases
contained over 40 different metrics. Using perl scripts, we ag-
gregated and grouped the complexity metrics (CBO, LCOM)
and the number of files. Each metric was only calculated from
C++ files. Each metric was calculated from a sub-system level
and the system overall. Another tool, SLOCCcount [24] was
also run over the raw source code for each release to get the
total number of lines contained in the C++ files.

Custom scripts were used to gather churn data on the git
repositories of both projects by looking at all the git logs
between each release. The scripts collected the number of C++
lines added and deleted. Release dates were determined by
looking for the version tag dates for each project.

To collect metrics for uniqueness, Git by a Bus [5] was run
over the git repositories to extract uniqueness metrics for all
developers in each release.

4) Step 4: In this final step, we had all the metrics in
excel sheets, but had run several calculations on the data,
and create graphs that would allow us to compare projects.
Using the R tool, we calculated correlations, Mann-Whitney-
Wilcoxon tests, and entropy. Entropy of unique knowledge
was calculated using entropy [14] package. In addition to the
metrics, we also looked at any process information that was
available, and checked README.txt notes for anything that
might explain different comparisons.
V. FINDINGS

In this section we provide a summary of the data collected and an interpretation of our findings to answer our original research questions. We begin first by examining the validity of the selected Lehman’s laws when applied to each project and then compare the evolution of both. As part of the comparison, we offer possible reasons to explain the differences based on our understanding of agile process and practices used by Chaste(agile) and BALL(non-agile). Metrics for both projects were collected for each release of the software, as defined by the team’s release notes and repository tags. In the figures below, both of the projects are graphed with the number of days since their respective first releases on the x-axis. Chaste(agile) has had fewer releases and operated over a shorter time interval.

For each of the metrics collected, we also ran Wilcoxon test on each of the datasets to determine if the distributions between Chaste(agile) and BALL(non-agile) were different. All of the Wilcoxon results indicate that the distributions of each dataset is different with a p-value <0.05 (see Table II).

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chaste LOC and BALL LOC</td>
<td>0.01554</td>
</tr>
<tr>
<td>Chaste CBO and BALL CBO</td>
<td>0.0008684</td>
</tr>
<tr>
<td>Chaste LCOM and BALL LCOM</td>
<td>0.0004456</td>
</tr>
<tr>
<td>Chaste churn and BALL churn</td>
<td>0.01154</td>
</tr>
<tr>
<td>Chaste entropy and BALL entropy</td>
<td>0.04342</td>
</tr>
</tbody>
</table>

A. Growth and Change

Figure 2 shows the graph of the SLOC for both Chaste(agile) and BALL(non-agile), which we use to assess Lehman’s sixth law of continuing growth. The graph in Figure 2 includes trend lines to measure how quickly each project is growing. Chaste(agile) is growing over twice as quickly as BALL(non-agile), with an increase of 17,387 lines added per release compared to 7,497 lines added per release of BALL(non-agile). BALL’s(non-agile) six most recent releases are a mix of major and minor versions, which explains the decrease in growth rate during some of the intervals. Based on this data, we conclude that Lehman’s first law of Continuing Growth applies to both projects.

When collecting measurements for growth, we included the subdirectories for test code that existed within both projects. Though it never reaches the end user, from an evolutionary perspective, the testing code is still an important part of the system and contributes to the overall maintainability of the code. The inclusion of test files had a direct impact on the differences in growth. While both projects included a test directory and encouraged their developers to write adequate tests, Chaste(agile) has a stronger emphasis on this practice. The Chaste(agile) development team practices test-driven development, which means they produce a set of automated tests for every feature implemented, which contributes to a high rate of growth, even with shorter release cycles [20].

Code churn was used to measure the law of Continuing Change. Figure 3 shows the log scale of code churn data we collected. In general, Chaste(agile) showed a higher degree of churn, which is expected because of the higher growth rate. We know the Chaste(agile) team incorporates refactoring as part of their implementation process, which we believe to be responsible for high churn rates, particularly when there is a lower amount of growth. For example, looking at the 4th release of Chaste(agile), we see a spike in churn, but there is very little change in the SLOC for that release compared to the previous release. The lack of similar patterns in BALL(non-agile) suggests to us there are some differences in the way that increased churn affects the project overall. While we believe this is the result of refactoring activities as part of agile development, more qualitative analysis is necessary to confirm this relationship.
Laws of Continuing Growth and Change hold true for both projects. The agile project grew faster and had higher levels of code churn, we believe because of consistent iterative development schedules and refactoring practices.

B. Complexity

Complexity was measured through two different metrics: coupling between objects (CBO) and lack of cohesion metrics (LCOM). The CBO was calculated for each C++ file and then an average was calculated for each release overall. Figure 4 shows the CBO level for each project. It is interesting to note that Chaste’s (agile) CBO grows at a consistent rate and is the higher overall for all releases. BALL (non-agile) has a lesser degree of coupling but a higher degree of variability, particularly in more recent releases.

Fig. 4: Law of Increasing Complexity measured by CBO per release.

Classes with higher LCOM numbers indicate that the responsibilities of that class are not well contained within it, and it should be subdivided into two or more classes. Classes with higher CBO indicate they are highly interconnected and related classes are accessing the internal details of one another. An ideal project would have a low degree of coupling and a high degree of cohesion, but in our case, the project with lower coupling has lower cohesion and vice versa.

In both projects, our metrics indicate complexity increasing over time, showing support for Lehman’s second law of Increasing Complexity. BALL (non-agile) has more variability in its metrics and shows a steady decline for both metrics for the first half of its releases. This is interesting as it is the direct opposite of what Lehman’s law would predict. Data collected by reviewing release notes suggests that BALL (non-agile) developers are able to decrease their complexity through the use of intermediate minor releases and focusing on maintenance during those periods. After the period of decrease for BALL (non-agile) there is an upward spike in both CBO and LCOM. Release notes shows at that time they added an experimental “linear algebra library” and added in “various convenience functions in core classes” [1]. The experimental library might not have been fully developed and could have had some design flaws. Two of the subsystems (MATH and QSAR) showed a significant increase in size (9,442 to 15,110 LOC and 4,872 to 14,945 LOC respectively) and a third (XRAY) was added to the project during that release.

We compared the complexity values using Wilcox tests and determined the distributions for CBO and LCOM were both significantly different between both projects (p < 0.01). The rate of change for both complexity metrics is increasing faster for Chaste (agile) indicating it is becoming more complex in a shorter amount of time when compared to BALL (non-agile).
Law of Increasing Complexity holds for both projects. The evolution of complexity shows statistically significant differences and the agile project’s complexity increases at a faster rate, we believe caused by fewer designated maintenance releases.

C. Familiarity

To assess developer familiarity, we used the “Git by a Bus” tool [5] to get a snapshot of the level of unique knowledge for each developer in every release in both repositories. The output of the tool lists each developer on the project along with a visual representation of their unique knowledge about the system. This is calculated by comparing code commits and looking for lines that overlap between different developers (i.e. if two or more developers edited the same line at any point after its creation). Figures 6 and 7 contain the visual output for the top developers on both projects for their most recent releases with the numbers representing knowledge points as calculated by the “Git by a Bus” tool [5]. In order to actually compare the familiarity of the two projects, we calculated the entropy using the top 100 developer’s unique knowledge. Figure 8 shows the entropy comparison between the two projects for each release.

Figures 6 and 7 show both projects having single developers with a substantial amount of unique knowledge compared to the rest of their co-developers. In the case of BALL(non-agile) (Figure 6) there is a single developer (Oliver) with nearly three times the number of unique lines as the next developer in the list. The level of uniqueness for each developer steadily drops moving down the list, which continues beyond the bottom of the figure. After the top 15 (displayed here), there are many more developers with very low levels of unique knowledge.
In the Chaste(agile) project (Figure 7) we can see a larger number of developers with top levels of unique system knowledge. There also appear to be two top tiers with 3-5 developers each having near the same amount of unique knowledge before it drops to a listing of those with less knowledge. However, an interesting point to notice is most of these “devs” with lesser knowledge actually represent pairs of developers with shared unique knowledge of system components. These pairs of people have a greater joint unique insight into the system than most of the miscellaneous individual developers on the system.

To understand how unique knowledge of developers changes over time, we decided to calculate the entropy of the unique knowledge held by single and multiple developers as given by the top 100 results by “Git by a Bus” [5] in each release of Chaste(agile) and BALL(non-agile). We want to indicate here that the top 100 results does not mean there are 100 developers working on the project, but 100 combinations including individuals and groups of two or more. When thinking about developer familiarity, we would want no developers to have unique knowledge, and would prefer all developers to know about every line of code. We want the amount of unique knowledge to be distributed between developers, and would prefer 3 developers having 50 lines of unique knowledge rather than one developer having 150 lines of unique knowledge, and the other two having no unique knowledge. Higher entropy values in our case would indicate the unique knowledge is more evenly distributed and spread out. A low entropy value would indicate that more of the knowledge is contained in one developer, thus more predictable.

Based on Figures 6, 7, and 8 we believe Chaste(agile) conserves familiarity of the system better than BALL(non-agile). Chaste’s(agile) process model puts a heavy emphasis on pair programming which would spread the unique knowledge. Though the developers work individually at times, they all participate in a pair programming session held once per week. During that time, all of the source code is pair programmed and pairs are rotated frequently with the intention of having every core developer on the team participate in the creation of every feature in some way [20].

Law of Conservation of Familiarity holds for both projects. Chaste(agile) displays better conservation of familiarity and a better distribution overall than BALL(non-agile), we believe because of strong pair programming practices.

VI. Threats to Validity

Our biggest assumption as part of this project is that we were indeed comparing an agile and non-agile project, the latter of which we were unable to concretely confirm. We selected Chaste(agile) specifically because of work published explaining its agile process [20], but there was nothing in BALL's(non-agile) documentation to suggest any process either way. We were able to get a response from BALL's(non-agile) development team which confirms our suspicion of no formal process being implemented. Particularly for BALL(non-agile), we were impeded by a lack of insight into their development practices. Future work in this area would benefit from an inside connection into the team and more opportunity for qualitative analysis.

To measure the complexity of the projects across time, we chose to look at CBO and LCOM metrics, and in retrospect, it would have been interesting to use cyclomatic complexity as a way to measure the complexity along with CBO and LCOM across the projects. We chose to use CBO and LCOM based on previous work by Singh and Singh [22]. Future work in this area may benefit from looking at all three metrics as a way of understanding the complexity of the projects.

With both projects, there was very limited bug reporting, making it difficult to assess the quality of the code. BALL(non-agile) has had a total of 85 defects reported [1] and Chaste(agile) has only 3 defect tickets reported [3]. We believe that the smaller user base of these projects is a factor that impacts the number of defects reported for each project. The lack of users may also have affected the other aspects of the code, but to our knowledge there is not much research correlating the number of expected users with the quality of work produced by the development team.

Throughout the project we encountered difficulty working with automated scripts to parse code files and commit histories. We chose to deliberately ignore any files that were not written in C++ by filtering out different file extensions in our automation, which could have caused to miss relevant files. Both projects have a significantly large number of commits, which prevented us from manually confirming the accuracy of our tools and scripts. At other points we were prevented from collecting the particular data that we wanted by restrictions enforced upon us by the tools at hand and our level of familiarity with their use.

There are two major factors that prevent our results from being generalizable across software projects. The first factor is...
our projects were developed in the open source community and has also been limited to mostly academic environments and it is possible that the results we found may only apply to open source and/or projects developed in academic environments. The second factor is the size of our data, we only used one project per development process (agile and non-agile) rather than comparing several of them indicating that the observations made may just apply to the projects we were looking at. A logical extension of this work would be to do an extensive study by looking at more data sets from several projects and see if the similar observations can be made. Despite these threats we still believe that our research and methodologies can be helpful in further studies.

VII. CONCLUSIONS

We looked at how the effects of some of Lehman’s laws differ between agile and non-agile open source projects in the scientific domain. The metrics we collected were used to support the laws concerning continuing change and growth, complexity, and familiarity. We found all the laws we were looking at still held true for both projects. We did find differences in the way the laws applied to the two projects. The agile project definitely grew faster, but also steadily increased complexity. By looking at the constant growth of the system we determined the law of familiarity held true for the agile project. Our data collection methods can be used in other research as a way of knowing what works and what does not. In future work, all of Lehman’s laws can be researched and determine their effects on an open source agile project. Another area of future work, is to instead comparing non-agile vs. agile, we could compare several similar projects using agile approach and finding out if they display similar evolution traits with regards to Lehman’s laws.

ACKNOWLEDGEMENTS

We would like to thank Dr. Emad Shihab for his assistance in providing valuable feedback towards this paper.

REFERENCES
