Characterizing Student Development Progress: Validating Student Adherence to Project Milestones

Bradley Erickson
North Carolina State University
Raleigh, NC, USA
bericks@ncsu.edu

Sarah Heckman
North Carolina State University
Raleigh, NC, USA
sarah_heckman@ncsu.edu

Collin F. Lynch
North Carolina State University
Raleigh, NC, USA
cflynch@ncsu.edu

ABSTRACT

As enrollment in CS programs have risen, it has become increasingly difficult for teaching staff to provide timely and detailed guidance on student projects. To address this, instructors use automated assessment tools to evaluate students' code and processes as they work. Even with automation, understanding students' progress, and more importantly, if students are making the 'right' progress toward the solution is challenging at scale. To help students manage their time and learn good software engineering processes, instructors may create intermediate deadlines, or milestones, to support progress. However, student's adherence to these processes is opaque and may hinder student success and instructional support. Better understanding of how students follow process guidance in practice is needed to identify the right assignment structures to support development of high-quality process skills.

We use data collected from an automated assessment tool, to calculate a set of 15 progress indicators to investigate which types of progress are being made during four stages of two projects in a CS2 course. These stages are split up by milestones to help guide student activities. We show how looking at which progress indicators are triggered significantly more or less during each stage validates whether students are adhering to the goals of each milestone. We also find students trigger some progress indicators earlier on the second project suggesting improving processes over time.

CCS CONCEPTS

• **Social and professional topics → Student assessment**; • **Software and its engineering → Software development techniques**; • **Applied computing** → Computer-assisted instruction.

KEYWORDS

CS2; Automated assessment tools; Progress indicators

ACM Reference Format:
Bradley Erickson, Sarah Heckman, and Collin F. Lynch. 2022. Characterizing Student Development Progress: Validating Student Adherence to Project Milestones. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2022), March 3–5, 2022, Providence, RI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3478431.3499373

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCSE 2022, March 3–5, 2022, Providence, RI, USA.
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9070-5/22/03…$15.00
https://doi.org/10.1145/3478431.3499373

1 INTRODUCTION

In computer science courses, projects are typically presented to students as a set of requirements, which they must implement. Evaluation of student work may include testing, both student-authored tests and teaching staff (TS) tests. To transform a set of requirements to code students must apply multiple cognitive and metacognitive skills such as problem decomposition and goal setting [5] as well as core software development practices including code design, test development, and debugging. This combination of skills may be overwhelming to novices [8]. Instructors will often support students' development processes by adding intermediate milestones with earlier deadlines that break the overall project into stages. Some examples include skeleton code, specific components such as an object hierarchy, or unit tests. This staging helps encourage the students to make useful progress rather than waiting until the last minute [16]. However the benefit of such approaches is limited by the rapid growth in CS enrollment [9], and by complex projects, which make it difficult to offer useful intermediate guidance. Checking each student's solution is simply infeasible in a large class. Instructors must cut down on the amount they assess by hand and thus, instructors have turned to automated assessment tools to evaluate intermediate and final submissions [6].

Using automated assessment tools along with version control systems such as GitHub allows us to collect data on students' development processes as well as their final performance [6, 7]. Edwards and Li [1], for example, analyzed process data to design a set of 15 progress indicators for code projects. Each indicator focuses on a single metric of productive activity, such as removing static analysis notifications or passing a previously failing unit test. In this work we analyze the evolution of these indicators across all students in a CS2 course over four stages of two projects. For each indicator, we determine whether or not there was a statistically-significant difference between the number of commits where the indicator was triggered during each stage.

In this paper we focus on the following research questions:

- **RQ1:** What types of progress indicators do students trigger during each stage of the project and how do they correspond to the expectations at each stage?
- **RQ2:** What differences exist in how the indicators are triggered between different projects either stage by stage or overall?

Our goal in this study is to gain a deeper understanding of how students make progress on their programming projects and how we can validate whether or not students adhere to the milestone expectations. Knowing the types of progress students make at each stage of a project can help TS tailor their feedback to help students be more successful and adjust future iterations of the course.

2 RELATED WORK

A central focus to our study is defining progress through a post-hoc analysis of progress indicators on student projects. In this section, we cover background on automated assessment tools which can support the live collection of progress indicators. Then, we discuss prior work in defining student project progress.

2.1 Automated Assessment Tools

As enrollments have risen, so too has the grading workload for instructional staff [14]. Instructors use automated assessment tools (AAT) to reduce time grading, so instructional staff can focus on student learning and help-seeking [6, 17]. One of the driving factors for using an AAT is the ability to offer students immediate feedback, typically written by the instructor, on their submission [14, 17]. If the feedback is useful, then the students are able to learn what they did incorrectly, correct the mistake, and re-submit for additional feedback. However, if the feedback provided by the AAT is not useful, then students may become discouraged in their ability to code [15].

While the general goal of an AAT is to automatically grade submissions, different AATs focus on different aspects of the process. In their literature review, Souza et al. [17] show that AATs have different assessment types, approaches, and specialties or customization. The customization aspect offers a path to guide students [6] or to collect the data for research purposes [11]. By collecting submission, build, and feedback data, there is opportunity to research how students make progress on their solutions [14].

2.2 Progress Indicators

Edwards and Li [1] took the usage of AATs a step further and proposed a set of 15 indicators for measuring student progress as they work on a solution using historical project data. The indicators are split into 2 categories: Indicators 1-7 are used for solution code and Indicators 8-15 focus on test code and the execution of the tests. Each indicator aims to capture when a single, potentially productive activity happens. To determine if any of the indicators are triggered, they compare the difference in build results between consecutive submissions by a student on the AAT Web-CAT for a given project. We define each indicator in Section 3.4.

These indicators have proven their usefulness in showing student progress and promoting a growth mindset through the use of recognition and encouraging feedback messages [2]. Additionally, Edwards and Li extend their work by using these indicators to gamify teaching non-course skills, such as time management or planning, by allowing students to complete achievements and level up based on their progress [3, 10].

3 METHODOLOGY

3.1 Course Background

The data for this study comes from the Fall 2020 offering of a CS2 course at North Carolina State University (NCSU) in Raleigh, a research intensive university in the mid-Atlantic United States. The course is the second of a three-semester introductory computer science sequence required for computer science majors and minors. The main topics of the course are software engineering practices (testing, coverage, static analysis, design), advanced OO (inheritance, interfaces, abstract classes), finite state machines, linear data structures (usage, implementation, testing, and runtime analysis), recursion, and recursive linear data structures.

Students in the course were expected to complete 2 projects, each of which are 22% of their final grade. Each project is split into two parts: 1) design and system test plan and 2) implementation and testing. For Part 1, students are expected to create a design proposal and system test plan for a set of requirements. For the more substantial Part 2, students are expected to implement a system designed by the TS; a UML class diagram and high-level function descriptions are provided. Implementation of a provided design allows for automated feedback and grading using a custom AAT [4]. Additionally, the TS writes unit test cases to evaluate the student implementation of the design. The source code for these tests are not provided to the students. For this study, we focus on Part 2 of the projects.

Students submit and manage their projects using a GitHub[1] Enterprise server. Each repository is associated with a job on the Jenkins[2] continuous integration system. When a student makes a new commit to GitHub, Jenkins pulls the repository, and runs an Ant-driven build process that compiles the student source and test code, runs static analysis tools (Checkstyle[3], PMD[4], SpotBugs[5]), compiles the TS test cases against the student code to ensure they are matching the public design, runs student-written tests instrumented for code coverage, runs TS test cases, and then provides feedback to the student about the build results. If the build fails during any of these steps, then the subsequent steps are not run. There are checks that may stop the build. If the student has a PMD notification for a poorly written test (e.g., `assertTrue(true);`), the TS tests will not run.

Part 2 of the projects included two internal deadlines, called *process points*, and a set of *done criteria* for the automatically graded portion of the project. The process points help the student maintain steady progress on the three week timeline for Part 2. For Fall 2020, we define the process points and done criteria as the following milestones:

- **Milestone 1 - Process Points 1:** Skeleton code that compiles, contains at least one test case, and is fully compatible with Javadoc (i.e. it raises no checkstyle notifications). Due about two weeks before the on-time deadline.
- **Milestone 2 - Process Points 2:** Student tests achieve 60% statement coverage on solution code. Due about one week before the on-time deadline.
- **Milestone 3 - AAT Done Criteria:** Student tests achieve 80% statement coverage, no static analysis notifications, all tests passing (student and TS). Students achieving the AAT Done Criteria see a green ball on Jenkins; however, they still

[1]https://github.com/
[2]https://www.jenkins.io/
[3]https://checkstyle.sourceforge.io/
[4]https://pmd.github.io/
[5]https://spotbugs.github.io/

have documentation refinement and system testing tasks to complete that are manually graded.

When a student achieves the 60% statement coverage, they 'unlock' feedback on the TS unit tests. Unlocking the TS unit tests allows them to see the number of tests that are passing or failing and for each failing test they receive a hint. This hint will give them a broad overview of the test, typically the test scenario, and the expected results. To receive full credit on the automatically graded portion of the projects, students must complete the final AAT Done Criteria milestone. Otherwise, students receive a percentage of the grade item related to how much they were able to accomplish towards the AAT Done Criteria.

While the process points had official deadlines where we evaluate the students' progress, the achievement of the milestone expectations is independent of the deadlines and typically achieved in the order provided. Thus we can define four stages of project development:

- **Stage 0:** Any commit before achieving Milestone 1
- **Stage 1:** Any commit between Milestone 1 and Milestone 2, inclusive of the commit that achieves Milestone 1
- **Stage 2:** Any commit between Milestone 2 and Milestone 3, inclusive of the commit that achieves Milestone 2
- **Stage 3:** Any commit after achieving Milestone 3, inclusive of the commit that achieves Milestone 3

The first project focused on implementing finite state machines and the students worked individually. For the second project, students had the option to work in pairs on implementing linear data structures.

3.2 Participants

Our IRB approval covers the use of student data with a waiver of consent. Students and their repositories were de-identified by replacing student emails and repositories with unique identifiers and removing all other student specific information. Since we are aggregating the progress indicators across all commits for all projects, individual student demographic data is not collected for this study. Table 1 shows we have a total of 495 repositories: 287 from Project 1 and 208 from Project 2.

3.3 Data Mining

The metrics for this study were collected after the semester concluded using BuildDataCollector[6]. BuildDataCollector iterates over repositories and runs a modified build for each commit. The build was modified from the one used for live evaluation to remove any blocking steps so we can mine all information about the student's submission at each commit. After each build is finished, BuildDataCollector gathers relevant build results from the generated output files. This includes commit metadata, notifications from static analysis tools, passing or failing student and TS test cases, coverage metrics from JaCoCo[7], and code counts from CLOC[8]. The data are stored in a MySQL database.

The BuildDataCollector collects raw build data. There are several calculated metrics that are used to identify if a progress indicator is triggered. We use the term 'src' to indicate a metric associated with student-implemented solution code. Any metrics associated with student-written test code is indicated with the term 'test'. The TS provides all user interface code, so user interface files are not included when calculating metrics. Most metrics are a sum of all the items of that type in the 'src' or 'test' portions of the code. For example, $testAsserts$ is the sum of all assert statements in the test code. The remaining calculated metrics are defined below. We gathered the $srcComplexMissed$ and $srcComplexCovered$ metrics through the complexity coverage measure from JaCoCo[18] as a measure of the cyclomatic complexity.

- $totalNotifications = count(Checkstyle) + count(PMD) + count(SpotBugs)$ for src and test code
- $srcComplex = srcComplexMissed + srcComplexCovered$
- $srcMethodSize = \frac{srcCode}{srcMethods}$
- $srcCommentDensity = \frac{srcComments}{srcCode}$
- $testSrcCodeRatio = \frac{testCode}{srcCode}$
- $testSrcMethodRatio = \frac{testMethods}{srcMethods}$
- $statementCoverage = \frac{srcCodeCovered}{srcCode}$
- $methodCoverage = \frac{srcMethodsCovered}{srcMethods}$
- $conditionalCoverage = \frac{srcComplexCovered}{srcComplex}$
- $assertDensity = \frac{testAsserts}{testMethods}$

After mining, we ran a script to calculate the progress indicators triggered and if milestone conditions are met, regardless of deadlines, for each commit.

3.4 Progress Indicators

The progress indicators (PI) rely on the metrics defined above generated from the mined data. To identify if an indicator is triggered, we compare the associated metric of the student's current commit to their immediately prior commit. Depending on if the difference in metric is positive or negative, the script records if the indicator has been triggered or not. Some indicators are triggered on a positive change (e.g., increasing statement coverage), while others are triggered by a negative change (e.g., decreasing test failures).

Our approach differs from how Edwards and Li [1] calculate the metrics. Instead of computing the metrics on a commit by commit basis, they keep track of the rolling maximum from the four previous commits. Additionally, we aggregate the number of commits an indicator triggers across all repositories for our analysis.

The progress indicators are split into two categories: 1-7 are for solution code and 8-15 focus on test code and the execution of the tests. We describe how we calculate each indicator:

Solution code indicators:

(1) **Increase solution methods:** The number of src methods, $srcMethods$, increase.
(2) **Removing static analysis errors:** The number of static analysis notifications, $totalNotifications$, decrease.
(3) **Reducing cyclomatic complexity:** The total src complexity, $srcComplex$, decreases.
(4) **Reducing average method size:** The average lines of code per method for src files, $srcMethodSize$, decreases.
(5) **Increase comments density:** The src comment to src code ratio, $srcCommentDensity$, increases.

[6]https://github.com/SOS-CER/BuildDataCollector
[7]https://www.jacoco.org/
[8]https://github.com/AlDanial/cloc

(6) **Increase solution classes:** The number of src classes, *srcClasses*, increase.
(7) **Increase correctness:** The student passes more TS test cases. Since we record the number of TS test failures, we identify when the student is failing less TS test cases, or *failures* decrease.

Test code and test execution indicators:

(8) **Adding new test methods:** The number of test methods, *testMethods*, increase.
(9) **Adding to existing tests:** The test code to src code ratio, *testSrcCodeRatio*, increases.
(10) **Increase number of tests per method:** The number of test methods compared to src methods, *testSrcMethodRatio*, increases.
(11) **Increase statement coverage:** The student-written unit tests cover more of their src code, *statementCoverage*, than it did before.
(12) **Increase method coverage:** The student-written unit tests cover more of their src methods, *methodCoverage*, than before.
(13) **Increase conditional coverage:** The student-written tests cover more of their conditional statements, *conditionalCoverage*, than before.
(14) **Increase assertion density:** The number of assert statements per test method, *assertDensity*, increases.
(15) **Increase number of test classes:** The number of test classes, *testClasses*, increases.

3.5 Evaluation

For each indicator, we conducted two proportion z-tests [13] across each pair of stages with Bonferroni correction [12]. The number of commits in each stage act as the populations, while the percent of commits where an indicator is triggered acts as the proportion. These tests tell us whether a significant difference (Bonferroni corrected p-value < 0.0083) exists between stages for each indicator. Next, we looked to see if any of the stages were significantly different from the other three. For each stage of the project, we look at which indicators happen the most or least often and compare them to the upcoming milestone deadline requirements. We conduct this process for each of the two projects and discuss the differences between them.

3.6 Threats to Validity

The data collected for this study comes from only a single semester, Fall 2020, of the CS2 course. The Fall 2020 semester was impacted by the COVID-19 pandemic. The pandemic caused a majority of the semester to be conducted online which may affect how students chose to work on their projects.

During the data mining process, before we conducted a full mine of the data available, we validated our results by manually assessing the data collected from a small set of student projects. Additionally, we kept logs of when data mining script ran into errors and resolved them on a case by case instance. The instructional staff used the same build process and procedures for each of the projects making the structure of the data similar which reduces threats to instrumentation.

Table 1: Summary statistics comparing the commit counts; the churn, or number of lines changed; the number of repos that finished each stage; and the total commits in each stage between the first and second projects.

	Project 1	Project 2	Overall
# Repos	287	208	495
Min commits	3	2	2
Median commits	67	85	74
Max commits	401	264	401
Avg. commits	72.4	100.8	84.3
Total commits	20,788	20,959	41,747
Min churn	0	0	0
25%	4	4	4
Median churn	12	12	12
75%	45	48	47
Max churn	4,256	4,428	4,428
Avg. churn per commit	62.3	58.5	60.4
# Repos finished in Stage 0	31	23	54
# Repos finished in Stage 1	6	3	9
# Repos finished in Stage 2	75	63	138
# Repos finished in Stage 3	175	119	294
Stage 0 total commits	5,931	6,177	12,108
Stage 1 total commits	1,382	1,658	3,040
Stage 2 total commits	12,392	12,477	24,869
Stage 3 total commits	1,083	647	1,730

4 RESULTS

Table 1 shows general commit; *churn*, or lines changed; and repository information for each project and overall. For the first project, we have 287 repositories with an average of 72.4 commits. The second project includes 208 repositories with an average of 100.8 commits. We did not find a significant difference in results when filtering out repositories with a large number of commits. The first project was completed individually and the second project was optional pairs, leading to fewer repositories. We notice that a majority of the commits had low churn; however, the average is dominated by a handful of large commits, for example committing TS provided UI code.

4.1 RQ1: Progress Indicators by Stage

For RQ1, we aggregated the commits for all repositories by stages. The bottom of Table 1 shows how many commits occurred in each stage for each project and overall. Most student work occurred in Stage 0, 29% of the overall commits, and Stage 2, 60% of the overall commits.

Table 2 shows the percentage of commits that triggered a given progress indicator for each of the stages for both projects. The italics indicates which of the proportions are significantly different from all other stages for a given indicator (p-value < 0.05).

4.1.1 Stage 0. During Stage 0 of both projects, the increasing solution methods (PI-01), reducing average method size (PI-04), adding new test methods (PI-08), increasing tests per method (PI-10), and increasing test classes (PI-15) indicators were triggered more than

Table 2: Percent of commits per stage that trigger each indicator. A "+" depicts an increase in each indicator, whereas a "-" depicts a decrease. Italicized values show an indicator proportion is significantly different than all other stages (p-value < 0.0083). Bold values highlight the stages each indicator is triggered more often in.

Indicator		Project 1 - Stages				Project 2 - Stages			
		0	1	2	3	0	1	2	3
PI-01	+ solution methods	***16.7%***	*4.5%*	0.9%	1.5%	***18.1%***	*5.0%*	2.0%	1.2%
PI-02	- static analysis notifcations	*17.2%*	***39.9%***	14.4%	11.7%	12.3%	***32.3%***	*14.8%*	10.2%
PI-03	- cyclomatic complexity	4.8%	**8.7%**	**9.1%**	5.3%	5.2%	***9.5%***	***11.8%***	4.0%
PI-04	- avg. method size	***18.3%***	11.8%	**15.1%**	13.3%	***21.4%***	10.0%	***17.4%***	11.1%
PI-05	+ comment density	22.0%	20.2%	17.9%	20.5%	***30.0%***	20.0%	22.6%	22.9%
PI-06	+ solution classes	*5.5%*	***9.3%***	1.1%	0.4%	*3.5%*	***8.6%***	1.9%	0.8%
PI-07	+ correctness	15.7%	**28.2%**	**30.1%**	12.9%	*15.6%*	***26.8%***	***37.0%***	*11.1%*
PI-08	+ test methods	**17.0%**	**17.7%**	7.5%	7.2%	**14.1%**	**14.2%**	7.3%	4.8%
PI-09	+ test code	*26.0%*	**37.0%**	**31.8%**	**32.9%**	23.4%	***42.6%***	***34.8%***	25.8%
PI-10	+ tests per method	***13.9%***	***20.8%***	8.3%	8.0%	***10.4%***	***15.5%***	8.0%	6.3%
PI-11	+ statement coverage	*16.1%*	*21.8%*	***39.6%***	***28.5%***	*14.2%*	***27.1%***	***35.4%***	*19.9%*
PI-12	+ method coverage	*8.8%*	***19.4%***	*13.5%*	*5.6%*	7.9%	***25.9%***	*12.6%*	6.2%
PI-13	+ conditional coverage	*13.1%*	21.1%	***29.4%***	21.3%	*12.3%*	**27.0%**	**27.9%**	*16.7%*
PI-14	+ assertion density	*13.1%*	***31.1%***	18.1%	17.5%	10.8%	***37.9%***	17.4%	13.4%
PI-15	+ test classes	***11.7%***	*8.5%*	0.9%	0.7%	***10.2%***	*8.0%*	1.6%	0.8%

they were in other stages of the project. These indicators are associated with the structure of the code rather than the functionality. The coverage indicators (PI-09, PI-11, PI-12, and PI-14) on the other hand were triggered less often in Stage 0. Since Milestone 1 requires a compiling skeleton and at least one test class, it makes sense that students are triggering the structural indicators and not the coverage indicators. While most indicators are similar for both projects, Project 2 triggers increasing comment density (PI-05) more often (30.0%) during this stage compared to Project 1 (22.0%).

Both PI-01 and PI-15 are triggered statistically more often in Stage 0 and decline thereafter. These indicators are related to skeletal pieces of code required for the first compilation milestone. While PI-08 and PI-10 are triggered often in this stage as well, they are not significantly different from all other stages. Each of these indicators are triggered more in Stages 0 and 1 relative to Stages 2 and 3.

4.1.2 Stage 1. Despite only including a small portion of the total commits (7.3%), Stage 1 saw several indicators being triggered more often than other stages: removing static analysis errors (PI-02), reducing cyclomatic complexity (PI-03), increasing solution classes (PI-06), increasing correctness (PI-07), adding new test methods (PI-08), adding to existing tests (PI-09), increasing tests per method (PI-10), increasing method coverage (PI-12), increasing conditional coverage (PI-13), and increasing assertion density (PI-14).

PI-02 is triggered significantly more often in this stage. This makes sense as the static analysis feedback is hidden from students until their code compiles, which is one of the conditions to pass Milestone 1. We still expect to see some commits trigger this indicator in Stage 0 since students are required to have zero CheckStyle notifications before completing Milestone 1. However, students may put off fixing notifications from the other static analysis tools until after they have achieved Milestone 1.

For PI-06, we may expect to see this commit follow the trend of PI-01 and PI-15 and be triggered more often in Stage 0; however, it is triggered significantly more often in Stage 1. Students are required to implement inner classes that are part of a state pattern for Project 1 and a linked list for Project 2 and they may hold off on creating these private inner classes until they have a compiling public skeleton.

We see a lot of commits trigger the testing and coverage indicators at this stage. PI-08 and PI-10 are triggered more often in the first two stages compared to the latter stages. PI-09, PI-12, and PI-14 are all triggered significantly more often in Stage 1. This implies the students are working on their test suite to meet the Milestone 2 coverage expectation.

Lastly, PI-07 is triggered more often in this stage compared to the first and last stage, but not Stage 2. Since the students are passing more tests, this implies they are making progress on their solution. PI-11 follows a similar trend to PI-07, but only for Project 2. We discuss this more in Section 4.2.

4.1.3 Stage 2. Stage 2 is where the bulk of the total commits happened (59.6%). In this stage, the indicators triggered the most often compared to other stages were: reducing cyclomatic complexity (PI-03), reducing average method size (PI-04), increasing correctness (PI-07), adding to existing tests (PI-09), increasing statement coverage (PI-11), and increasing conditional coverage (PI-13). The theme of these indicators are driven by testing, both TS and student-written tests, and the refinement of the solution code. We see indicators related to the increases in methods (PI-01 and PI-08) or classes (PI-06 and PI-15) triggered significantly less beginning in this stage. These reflect changes to the structure of the code, which we do not expect to see this far along in the projects.

For PI-07, PI-11, and PI-13, we see an increase in trigger rate leading up to the peak in Stage 2. Then, these indicators see a sharp decline in trigger rate during the final stage. These indicators are related to passing tests and achieving statement coverage, which are key elements to reaching Milestone 3 and entering Stage 3.

While PI-09 was triggered significantly more often in Stage 1, we still see many commits triggering this indicator in Stage 2, suggesting that students are refining or adding to their tests from feedback on TS test failures. PI-03 and PI-04 are indicators associated with refining solution code, which would likely happen as students work on fixing failures associated with TS unit tests.

4.1.4 Stage 3. Finally, Stage 3 saw the least amount of commits overall (4.1%). Again, we see the indicators related to structure are triggered much less (PI-01, PI-06, PI-08, and PI-15). Additionally, removing static analysis errors (PI-02), reducing cyclomatic complexity (PI-03), and increasing tests per method (PI-10) are all triggered less during this stage. Students in Stage 3 have already completed their test suite and removed all static analysis notifications per Milestone 3.

We do see PI-07, increasing correctness, triggered some in this stage, even though students should already be passing 100% of TS unit tests per Milestone 3. After manually checking the data, we notice that a few students achieve Milestone 3, and thus may move into Stage 3; however, while they are cleaning up their code before the final submission, they make a breaking change and need to fix their code. Thus, they trigger PI-07 when fixing the test failure.

There are only two indicators that happen more often compared to other stages. They are adding to existing tests (PI-09) and increasing statement coverage (PI-11). The students are not required to go beyond the 80% statement coverage threshold, but we noticed many students went above this.

4.2 RQ2: Progress Indicator Project Differences

For most of the progress indicators, the values and the trends were relatively similar across projects, we will highlight the differences between Project 1 and Project 2 here to answer RQ2.

During Project 1, fewer commits triggered the increasing comment density (PI-05) during Stage 0 when compared to Project 2: 22.0% for Project 1 and 30.0% for Project 2. Approximately one third of the students lost points on Project 1 from CheckStyle notifications relating to Javadocing their code. Project 2 saw fewer students miss the points associated with these notifications. This implies the students learned from their mistakes on the first project.

Additionally, there is a difference between projects related to increasing statement coverage (PI-11). Project 1 saw students focusing on statement coverage during Stages 2 and 3; whereas, Project 2 saw students focusing on statement coverage more during Stages 1 and 2. After Project 1, students may have realized the importance of statement coverage as a metric of evaluation and changed their focus to the metric earlier in the project.

Evaluating over both projects do not yield significantly different results to what we have already shown and thus, we excluded those for space.

5 DISCUSSION

Overall, the progress indicators demonstrate that students in the CS2 course are making the appropriate and intended progress during the stages of their software development projects. Since one of the goals of our project is to encourage best practices for software development, the patterns of progress indicators suggest that students are meeting those expectations.

However, the progress indicators, in their current form, only tell part of the story about student progress toward solution for programming projects. In particular, the solution indicators lack an indicator for when the solution code increases. The focus of the progress indicators is on perfecting almost complete student solutions rather than building up to a solution. Therefore, in Stage 1, the triggered indicators are related to improving the test portion of the code, but provide no details about the effort students are putting into the solution portion of the code. The intention of Stage 1 is that students are working on both solution and test, but students may focus on test to achieve coverage and hold on implementing solution until Stage 2. With the indicators in their current form, we are unable to check whether or not this unwanted behavior is occurring. Additional indicators may be needed to provide clarity on what might be forward progress in different stages of the project. During the start of the project, solution code should be increasing. Later in project, we would expect refinement related to reducing cyclomatic complexity (PI-03) and reducing average method size (PI-04).

Additional refinement of the indicators to measure different types of progress at various stages of software development would then allow us to support new feedback mechanisms. For example, if students were not implementing solution code in Stage 1, the build process could include feedback with suggestions on where to focus effort for the project. The majority of work occurs in Stage 2, so an adjustment of expectations as measured by progress indicators, may help balance the workload for students.

Edwards and Li [1] identified that commits or submissions that triggered six of the 15 indicators suggested that students were making progress on a programming assignment as part of their evaluation on the suitability of the progress indicators for use. The assignment considered, to our knowledge, did not have stages as defined in our study and therefore, the analysis of the indicators triggered at the start, middle, and end of the project was not considered. We build on the existing progress indicator work by considering how the indicators can support the identification of positive student behaviors when completing programming tasks across the development timeline.

6 CONCLUSION

We observed patterns in the type of progress students make that reflect the expectations for each milestone. We saw this with skeletal progress in Stage 0, statement coverage progress in Stage 1, and test-focused progress in Stage 2. Additionally, we saw students commenting their code more often in Stage 0 of Project 2 and the students started working on coverage earlier in Project 2's life cycle compared to the first project suggesting improved development processes.

The next steps for this research are to validate our results against other offerings of the same course that have similar milestone specifications and compare student performance to the indicators being triggered at each stage. Additionally, we can compare our results with course offerings that contain different milestone definitions or different types of feedback mechanisms.

7 ACKNOWLEDGEMENTS

This material is based upon work supported by NSF under grants #1525173 (PI: Sarah Heckman), #1714538 (PI: Sarah Heckman), #1821475 (PI: Collin F. Lynch, CO-PI: Tiffany Barnes and Sarah Heckman), and #1934975 (PI: Sarah Heckman, CO-PI: Anna Howard and Tzvetelina Battestilli).

REFERENCES

[1] Stephen Edwards and Zhiyi Li. 2016. Towards Progress Indicators for Measuring Student Programming Effort during Solution Development. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '16)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/2999541.2999561

[2] Stephen H Edwards and Zhiyi Li. 2019. Board 43: Designing Boosters and Recognition to Promote a Growth Mind-set in Programming Activities. In *2019 ASEE Annual Conference & Exposition*.

[3] Stephen H. Edwards and Zhiyi Li. 2020. A Proposal to Use Gamification Systematically to Nudge Students Toward Productive Behaviors. In *Koli Calling '20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '20)*. Association for Computing Machinery, New York, NY, USA, Article 28, 8 pages. https://doi.org/10.1145/3428029.3428057

[4] Sarah Heckman and Jason King. 2018. Developing Software Engineering Skills Using Real Tools for Automated Grading. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 794–799. https://doi.org/10.1145/3159450.3159595

[5] Cindy E Hmelo-Silver. 2004. Problem-based learning: What and how do students learn? *Educational psychology review* 16, 3 (2004), 235–266.

[6] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '10)*. Association for Computing Machinery, New York, NY, USA, 86–93. https://doi.org/10.1145/1930464.1930480

[7] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, Miguel Ángel Rubio, Judy Sheard, Bronius Skupas, Jaime Spacco, Claudia Szabo, and Daniel Toll. 2015. Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports* (Vilnius, Lithuania) *(ITICSE-WGR '15)*. Association for Computing Machinery, New York, NY, USA, 41–63. https://doi.org/10.1145/2858796.2858798

[8] Ioannis Karvelas, Annie Li, and Brett A. Becker. 2020. The Effects of Compilation Mechanisms and Error Message Presentation on Novice Programmer Behavior. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 759–765. https://doi.org/10.1145/3328778.3366882

[9] Kathleen J. Lehman, Julia Rose Karpicz, Veronika Rozhenkova, Jamelia Harris, and Tomoko M. Nakajima. 2021. *Growing Enrollments Require Us to Do More: Perspectives on Broadening Participation During an Undergraduate Computing Enrollment Boom*. Association for Computing Machinery, New York, NY, USA, 809–815. https://doi-org.prox.lib.ncsu.edu/10.1145/3408877.3432370

[10] Zhiyi Li and Stephen H Edwards. 2020. Integrating Role-Playing Gamification into Programming Activities to Increase Student Engagement. In *2020 ASEE Virtual Annual Conference Content Access*.

[11] Raymond Lister. 2010. CS EDUCATION RESEARCH<br><br>The Naughties in CSEd Research: A Retrospective. *ACM Inroads* 1, 1 (March 2010), 22–24. https://doi.org/10.1145/1721933.1721942

[12] Ron C Mittelhammer, George G Judge, and Douglas J Miller. 2000. *Econometric foundations pack with CD-ROM*. Cambridge University Press.

[13] Roxy Peck. 2008. *Introduction to statistics and data analysis*. Thomson Brooks/Cole, Australia Belmont, CA.

[14] Raymond Pettit and James Prather. 2017. Automated Assessment Tools: Too Many Cooks, Not Enough Collaboration. *J. Comput. Sci. Coll.* 32, 4 (April 2017), 113–121.

[15] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive Difficulties Faced by Novice Programmers in Automated Assessment Tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Espoo, Finland) *(ICER '18)*. Association for Computing Machinery, New York, NY, USA, 41–50. https://doi.org/10.1145/3230977.3230981

[16] Clifford A. Shaffer and Ayaan M. Kazerouni. 2021. *The Impact of Programming Project Milestones on Procrastination, Project Outcomes, and Course Outcomes: A Quasi-Experimental Study in a Third-Year Data Structures Course*. Association for Computing Machinery, New York, NY, USA, 907–913. https://doi-org.prox.lib.ncsu.edu/10.1145/3408877.3432356

[17] Draylson M. Souza, Katia R. Felizardo, and Ellen F. Barbosa. 2016. A Systematic Literature Review of Assessment Tools for Programming Assignments. In *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*. 147–156. https://doi.org/10.1109/CSEET.2016.48

[18] Arthur Henry Watson, Dolores R Wallace, and Thomas J McCabe. 1996. *Structured testing: A testing methodology using the cyclomatic complexity metric*. Vol. 500. US Department of Commerce, Technology Administration, National Institute of ....