UCAT: The Unified Codio Autograding Tool

by

Joel G. Bland

B.S., Kansas State University, 2014

A REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2023

Approved by:

Major Professor
Josh Weese

Copyright

Abstract

To meet increasing enrollment demands in computer science courses, many institutions employ automated coursework assessment tools. In this report, we present such a tool titled UCAT: The Unified Codio Autograding Tool. Codio is a web service for administering programming homework assignments with only limited grading capabilities. UCAT expands these capabilities using unit-test-based assessment for partial credit, a feedback pipeline from instructor to student, automatic static code analysis, and more. UCAT is designed to be platform-independent, unifying assessment efforts across different programming languages into one tool.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I wish to thank Dr. Josh Weese and Russell Feldhausen for the opportunity work on a tool that will aid current and future educators. Thank you both for the project guidance, debugging help, and support throughout my education!

Thank you to Russell for providing Version 0 of UCAT, which was an excellent starting point.

Thank you to colleague, classmate, and friend Anthony Atkinson, who recommended the tool ANTLR to me when I was attempting to reinvent Java parsing and lexing in Python.

Thank you to my partner Alicia, who, on top of going above and beyond by helping me with chores, errands, and cooking, has patiently waited for me to finish this academic excursion. Everything is easier with you.

Dedication

For my parents, Drs. Paul and Lendi Bland. They are lifelong educators and I am proud to call them Mom and Dad.

Chapter 1

Introduction

In computer science education, programming assignments are crucial to the curriculum.[1–3] However, in the last twenty years, computer science course enrollment has seen constant increases that have often outpaced teaching resources.[4–7] Analyzing student code and providing constructive feedback for programming assignments requires significant time and effort from instructors,[1;2;8] but modern class sizes render the manual assessment of individual submissions infeasible.[3;6;9] The solution many institutions turn to is the employment of automated assessment tools,[4;5;7–11] often referred to herein as *autograders*.

Used effectively, autograders confer many benefits to instructors and institutions with the chiefest and most self-evident among them being savings in instructor time, effort, and resources.[1;5;12] These economic savings along with the scalability of some systems lend themselves to larger class sizes,[5;12] allowing institutions to meet the increasing enrollment demands. Furthermore, there is evidence that autograders can improve student learning outcomes[5;12;13] for a variety of reasons, such as increasing the possible amount of assigned programming tasks,[1] maintaining higher levels of student self-motivation,[7] and improving the student experience.[7]

Here we present such a tool titled UCAT: the Unified Codio Autograding Tool. Codio[14] is a web service for developing, administering, and assessing programming homework assignments. At time of writing, this service provides only rudimentary autograding capabilities

limited to program input-output comparisons, resulting in pass/fail grades. Using the Codio environment, UCAT expands the autograding ability by offering partial credit via unit test suites, providing a pipeline for automatic hints and guidance, and generating a Unified Modeling Language[15] (UML) class diagram image of the student's code submission. Moreover, UCAT was designed so that its core operation and architecture are independent of the assessed submission's programming language; hence, it unifies assessment efforts across different programming languages into one tool.

This report continues in Chapter 2 with a review of research literature related to autograders, where topics discussed include autograder history, pedagogy, functional requirements, and a discussion on prior work. Chapter 3 then provides in-depth technical documentation of UCAT, including its origin, features, architecture, usage, and limitations. Finally, Chapter 4 concludes the report and describes future work.

Chapter 2

Literature Review

UCAT is one of many tools in a long history of automatic assessment of coursework in computer science education (CS-Ed). This chapter covers the origins of computer-aided assessment in CS-Ed, related teaching practices, required features, and a discussion regarding potential redundancy of work.

2.1 Historical Origins

Autograding tools are an old concept in CS-Ed, having been used as early as 1959. At the time, programming exercises were assigned in assembly language and written using punch cards which would then be fed into a computer programmed to assess the assignment correctness. Interestingly, even this early tool made possible a distance education program in which students could send their punch card assignments through the mail to be assessed by the autograder machine on campus.[12]

Douce et al. offer a timeline of the progression of autograders in their 2005 review of automatic assessment tools for programming assignments. This timeline spans the 1960s through the early 2000s, distinguishing three main eras. The first era is that of early computing courses in the 1960s and 1970s, where the creation of autograders required "a great deal of expertise." These tools had basic features such as checking program correctness, mon-

itoring program efficiency, and maintaining a grade book. As technology progressed in the 1980s through the 1990s, a second "tool-based" era was entered in which autograders began to take the form of utility programs which were often scripted in nature. This era saw the advent of command line and graphical user interfaces for autograders, as well as further improvements to course management and administrative functions. Finally, as web technology saw its rise in the late 1990s and early 2000s, a third era began. Autograders began to use web-based platforms and further advances in technology allowed for even more sophisticated assessment methods, including unit test suites and an increased focus on providing more precise feedback.[2] Now, in 2023, web services for administering programming assignments are common, providing integrated development environments (IDEs), assignment specification tools, and automatic assessment tools in many different platforms.[10;14;16–20]

2.2 Autograder Pedagogy

While autograders can streamline operations for a large course, they must be used with care. Poorly implemented systems can negatively impact student learning, so multiple pedagogical aspects must be considered when employing an autograder. In this section, the topics of assessment methods, feedback, and student behavior are discussed as they relate to autograders.

2.2.1 Assessment

Assessment of student work plays a critical role in education. The results of assessment provide information on the educational process to both students and teachers, and allow teachers to monitor learning outcomes from perspectives ranging from course-level to student-level. Assessment results can also guide student learning and inform the nature of feedback that should be presented. In the context of programming courses, automated assessment tools facilitate continuous assessment, encouraging sufficient programming practice by students.[8]

To properly assess student work, a set of criteria must be established to determine how

well a submission fulfills the requirements. Joy et al. state "there appears to be no agreed set of criteria" to measure the quality of a computer program; however, they do present a model that formulates a generalized assessment method for programming assignments. Fig. 2.1 shows this tree.

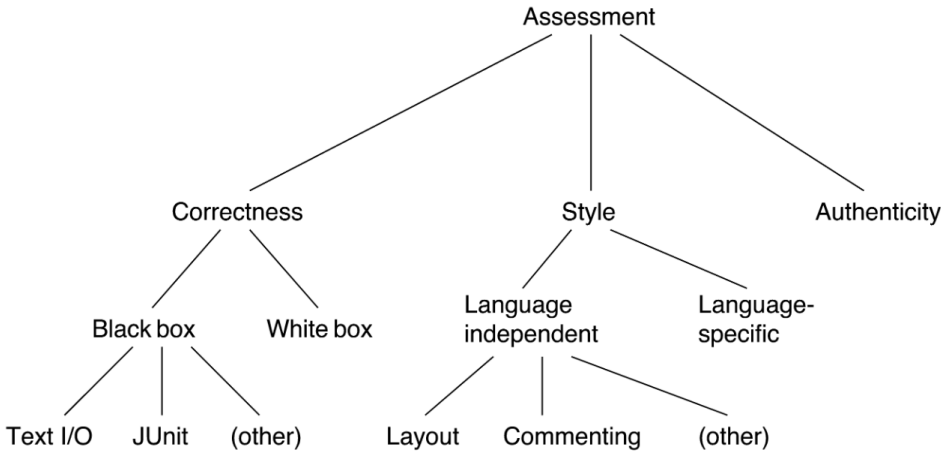

Figure 2.1: *Assessment Tree for Programming Assignments*[21]

The authors determine that the three main components of programming assignment assessment are correctness, style, and authenticity. Assessing the correctness of an assignment involves checking that it fulfills functional requirements set by the instructor, while style assessment looks for proper code writing conventions. Authenticity checks ensure no plagiarism has occurred and verify the identity of the student for proper records. Correctness checks are then delineated by separating them into white box and black box testing techniques, which respectively test the code with knowledge of its contents and without such knowledge. Style checks are separated into categories of language-independent and language-specific conventions.[21]

Some aspects of these components are well-suited to automation while others are not.[1;21] For example, Joy et al. state that it is difficult to automatically assess inclusion of code comments and effective usage of a given programming language's features. To judge these, a human assessor would likely do better, and multiple authors acknowledge that in the ideal case a mixture of manual and automatic assessment would be employed, especially as assignments increase in complexity.[1;8;21] Given the acknowledged effort required for such tasks,[1;2;8]

it follows that autograders should focus on assessing the aspects that can be fully automated such as program correctness and simple style requirements.[1] One of the most common methods for automatically assessing correct functionality is running a predefined unit test suite against the submission.[1;2;8;22] Another strategy used in the past is the comparison of program inputs and outputs to expected values, but this has long proven to be an inferior method of assessment for many reasons.[13;21–24]

Automated assessment must not only reduce instructor burden, but also help students learn. A fundamental requirement is that students understand what specific qualities of a submission the autograder will assess, and so the programming assignment specifications must be carefully created and presented to the students.[2;25] Student acceptance of autograders is important to their successful utilization,[26] so in order to prevent frustration, discrepancies between specification and autograder results should not exist.[23] On the other hand, overly restrictive specifications are to be avoided, as they can stifle student creativity, which is important to encourage in computer science and education in general.[21;23;27]

2.2.2 Feedback

Presenting feedback to students is one of the most important practices in education, and there is a deep well of research on how best to provide it to learners in the general case.[25;28;29] Both Shute and Ott et al. provide extensive literature reviews on the topic and discuss the nature of effective feedback. Shute states that effective feedback has two parts: verification and elaboration. As the terms imply, verification confirms whether an answer is correct, and elaboration then provides more information on why it is so, or otherwise helps a student revise their answer. Ott et al. also state that effective feedback advises students on their performance as well as ways to improve. They say feedback should be "regular, detailed, and timely."[25]

Shute's literature review focuses on a type of feedback called formative feedback, which they define as "information communicated to the learner that is intended to modify his or her thinking or behavior for the purpose of improving learning." They advocate for this spe-

cific type of feedback, detailing many benefits like reducing student uncertainty regarding their own performance, reducing student cognitive load, and correcting improper task approaches and misconceptions. This type of feedback may be particularly suited to automatic assessment.[30]

Given the importance of feedback in education and the subtle variations that may affect learning, feedback delivered by automated assessment tools must be designed with intention. One of the main benefits of feedback delivered by such tools is its immediacy; the ability for students to get immediate feedback on their submission is considered to be especially effective for programming assignments.[28;31] However, generating the content of the feedback presents a greater challenge. Feedback may be more effective when it tends toward specificity rather than generality, but the complexity of the information presented as well as the intended audience must be considered.[25;28] Students may disregard feedback that is too long, too detailed, or too complex for their current aptitudes,[25;28] so technical feedback should be written for the typical level of experience for whom it is assigned. Autograder feedback is commonly compared to error messages generated by compilers or IDEs, which novice students especially have difficulty interpreting because these messages are often vague, highly technical, or contain jargon.[13] With this in mind, feedback designed for an autograder should at minimum consist of the types of errors observed as well as their nature; more critically, the feedback should contain advice on how to proceed.[22;25] Finally, the feedback should present a measure of the extent to which the submission was accepted to inform students how well they performed the task.[10] A challenge for automated feedback is that individuals respond to various types of feedback differently. In an ideal case, feedback is delivered in a manner that best suits the individual student,[25] but this is at odds with the nature of automated tools designed for large courses.

2.2.3 Student Behavior

It is thought that the introduction of an automated assessment tool to a course can change the way students approach the coursework.[8] Student behavior should then be a consideration

in the design and application of autograders. To help students learn, autograders must be psychologically acceptable to them. Ideally, students will like using the tool, but at a minimum they should see it as a legitimate authority in delivering marks.[1;2;26] It has been shown that student attitudes toward an automated assessment tool may affect the tool's ability to aid learning; specifically, the tool may be less helpful for students who view it unfavorably.[26] Unfavorable views often stem from functional or design limitations such as insufficient feedback or grading procedures perceived as unfair,[2;13;24;32] so aspects such as these must be considered in automated assessment tool design.

If used improperly, autograders can also reinforce undesirable learning behaviors. Computer science students tend to focus on whether their program outputs are correct and disregard other aspects like style and algorithm design,[21;31] so automatic grading may discourage deep thinking about the material. In fact, it has been seen in many cases that students often come to rely on the autograder results as an indication of submission acceptability without internalizing the lesson at hand.[1;5;8;11;31] For these reasons, unlimited autograder submissions are considered detrimental to learning. However, autograders are meant for on-demand assessment, and allowing students multiple attempts to receive results before a deadline can be beneficial.[5;31] The compromise is to allow multiple submissions, but limitations or penalties must be imposed to prevent over-reliance on the tool.

2.2.4 Summary

When it comes to assessing coursework for large student cohorts, autograders certainly solve the problem of scale. Yet, the details of the implementation must be carefully considered to facilitate student learning alongside the easing of instructor burdens. Requirements for automatically assessed assignments must be adequately specified so that students understand what the tool is assessing, but they must not be specified to such an extent that student creativity is dampened. Feedback must be designed with student aptitude and desired learning outcomes in mind. It is helpful to allow students multiple assignment submissions for which they receive immediate feedback, but allowing unlimited submission attempts can have neg-

ative effects on learning. In all, it seems that good autograder design treads a thin line and may require fine-tuning to achieve the best results.

2.3 Functional Requirements

Based on research literature related to the pedagogical implications of automated assessment tools, multiple functional requirements for autograders have been identified. The main function of an autograder is to determine some measurement of a submission's quality as compared to a standard set by the instructor;[1] in doing so, it should utilize the following features.

2.3.1 Secured Environment

The autograder must assess a programming assignment in a secured environment. It must protect against potentially damaging side effects of executing student code, whether caused by bugs or intentionally malicious features.[1;8;10;12;21] It has been recommended to provide students with an isolated development and assessment environment, often called a sandbox, for such reasons.[1;10] Autograders must also maintain the privacy and integrity of students who use them. There should be no risk of a student gaining access to a grade book or the model solutions. The students must be unable to falsify autograder results, either by modifying the model solutions or other means.[10] Additionally, there should be plagiarism checks in place.[21]

2.3.2 Functionality Assessment

The autograder must check to what extent the submission achieves the requirements determined by the instructor.[1;2;8] A prerequisite capability then is that the autograder must be able to compile or interpret the student code, depending on the language. This verifies the code contains correct language syntax.[1;10] Once syntax is verified, there are many ways to assess required functionality, but the most common strategy is the execution of a unit test

suite defined by the instructor.[1;2;8;22]

2.3.3 Meaningful Feedback

The autograder must provide feedback that helps students learn. An autograder should at minimum both clearly state the results of the assessment and give guidance regarding any errors or failings detected.[10;22] Though unit test suites are among the most common methods of automated functionality assessment,[1;2;8;22] Keuning et al. point out that "test-based feedback will not in all cases help a student to fix an incorrect program." Thus, a variety of feedback sources should be considered, such as insights from methods such as static code analysis.

2.3.4 Static Code Analysis

The autograder should perform some form of static code analysis. Static code analysis is done by examining the code without executing it, which can reveal problems not detected by unit tests, assess programming style and conventions, and check for adherence to design and architecture requirements.[1;10;33] Though this type of assessment yields important feedback for students, it may be difficult to fully automate.[1;21]

2.3.5 Discouragement of Autograder Over-Reliance

The autograder should include a means of managing students' reliance on its results as an indication of the fulfillment of assignment requirements or lesson mastery. Simple methods of discouragement include limiting the total quantity of allowed executions or enforcing minimum wait times between them,[8] but Baniassad et al. propose a more active means of discouragement. They recommend applying a penalty to the final grade each time the autograder returns a lower score than in previous attempts.

2.4 On Potential Redundancy of Work

Autograders have existed for more than 60 years and many iterations of tools have been created in that time. For instance, in their 2018 literature review, Keuning et al. surveyed 101 different tools. Ihantola et al. speak at length on this aspect of autograders, questioning why so many tools exist, and in particular, why this work continues to be repeated. This quote seems especially apt:

> One clear reason for the variety of tools has to do with their availability and lifespan. Tools are often created as a part of a thesis or for a particular course. They are finished enough for studying a research question or to support the needs of one particular course, but are not suitable for distribution. It is rather common that the very first version of a tool was something that the teacher did quickly for his/her very own purpose. These tools might get publicized if some research was the original motivator, but as they never emerge as supported pieces of software, similar systems get implemented again and again. Correspondingly, there are far [fewer] systems that are widely adopted than there are papers about new tools.[8]

This quote is included here because it nearly perfectly describes the nature and origin of UCAT. It was originally quickly created by faculty as an internal tool to support the needs of a curriculum and then handed off for further development as the topic of a master's report. At time of writing, plans are such that the source code will be made publicly available, but it won't be supported as an open-source tool. Ihantola et al. go on to lament the frequent "reinvention of the wheel", but we offer a counter to this sentiment in the introduction to Chapter 3.

Chapter 3

The Unified Codio Autograding Tool

UCAT is an automatic grading tool that is written mostly in Python[34] and designed to be executed via Bash[35] script in an Ubuntu operating system[36] provided by the Codio web service. For a given assignment, instructors create sets of unit tests in the graded language, write feedback in JSON files,[37] and configure an initialization shell script for the tool. The Python portion of the tool parses the instructor inputs, assesses student work by calling language-specific unit test runners, calculates assignment scores, handles logging and student feedback, and reports the assignment score to Codio. This tool invokes operations in other languages via Ubuntu system commands. UCAT is a modern autograder that interfaces with a CS-Ed web service, assesses student code using unit tests and automatic static code analysis, and is feedback-focused with an instructor-to-student feedback pipeline.

As acknowledged in Section 2.4, there are many existing tools similar to UCAT. While "reinvention of the wheel" can inhibit progress in the state of the art generally, there are both practical and market reasons for an institution to create (and not necessarily publish) its own tools instead of using something in existence. First, many computer science educators possess the skills required to create their own autograder. A from-scratch project allows the creator to tailor it specifically to their needs and use case as we have done with Codio and UCAT. Additionally, using an existing open-source solution relies on the time and goodwill of its developers to maintain it. A tool maintained in-house doesn't have this dependency external

to the institution. Furthermore, an internally-created tool avoids the financial costs of a commercial product and allows the flexibility to change its functionality at will as pedagogical best practices change over time. Publishing an open source tool is also a commitment to the community at large in which instructors may not be willing to participate for reasons of time and effort. And finally, since automated assessment tools are reasonably complex and open-ended software engineering problems, they can further students' education as the focus of an implementation project. Indeed, the advantage of the current state of autograder literature is that when an institution sees a need for an internal tool, there is a wealth of prior work and discussion on the topic.

3.1 Origin

At Kansas State University, the Computational Core curriculum is a set of classes designed to teach students of all majors the fundamentals of computer science.[38] Codio was the tool chosen to administer programming assignments whose features include a web-based IDE, a sandboxed code execution environment, assignment specification tools, and gradebook functions. Codio's model is infrastructure-as-a-service, meaning that Codio's main offering is the technology and framework for instructors to configure their own assignments in an environment suited for doing so. For each assignment, Codio provides what they refer to as a Box, which is really a Docker container[39] running Ubuntu accessible through a web browser. An instructor account creates a master assignment in one of these Boxes and then student accounts access their own instance of it. Codio does not provide automatic assessment aside from a token program input-output comparison, but it does provide a software hook to initiate a shell script created by an instructor or developer for custom grading procedures. This was the inspiration for the predecessor to UCAT, created by K-State faculty.

UCAT's predecessor was created to leverage unit test capabilities within the two programming languages taught in Computational Core classes: Java[40] and Python. This approach led to different versions of the tool being created for the different languages, which was convenient for instructors focusing on one at a time, but the obvious drawback was that there

were multiple code bases to maintain. Furthermore, in early configurations, the automated assessment would error and terminate when student code did not perfectly match structural specifications, such as class names, even if functionality was correct. The solution was to use code reflection to compile the code independently from the unit test suites, mitigating this problem. Code reflection in unit test suites will continue to be used alongside UCAT, but since UCAT was designed to interface with externally-created unit test files, ensuring that the code reflection is properly configured is out of scope for UCAT developers and must be left to the instructors.

The first tool calculated the assignment grade as the percentage of unit tests that passed out of the whole suite. Feedback to students was fairly rudimentary, consisting of a hard-coded string associated with a unit test alongside the unit test assertion. These were done using the Hamcrest[41] unit test framework. Overall, this version achieved automated assessment of the coursework, but clear avenues of improvement existed.

3.2 Features

UCAT has a number of features that improve upon its predecessor. It exists as one tool, does static code analysis, expands and provides feedback customization, records student performance data, and more.

3.2.1 Platform Independence

UCAT is written in Python and uses separate modules with a standardized interface to handle each supported assignment language. Using an initialization shell script, an instructor tells UCAT which language to use. Within the specific language handler module, the graded language is invoked by the appropriate means. For example, for Python, it continues in the Python environment with PyUnit,[42] the Python unit test framework. For Java, it uses Ubuntu system commands to run a pre-configured Java wrapper project that runs JUnit,[43] the Java unit test framework. To add support for more languages, a developer simply needs

to add a module that conforms to the UCAT main body input-output specifications defined in Appendix B. Aside from those requirements, most anything can be done within the module in order to interface with the new language.

3.2.2 Automatic Class Diagram Creation

UCAT is able to take student code defining a class and automatically create a UML[15] class diagram image and present it back to the student. This is done using a toolchain of different libraries. To extract the relevant class information from the code, ANTLR (ANother Tool for Language Recognition)[44] is used. ANTLR is a parser generator that takes a context-free grammar written by a user in the ANTLR paradigm. It automatically generates parser and lexer files in a specified programming language to be used in interpretation of code written in such a grammar. The ANTLR organization provides grammars for many programming languages, and these resources were used to generate Java parser and lexer files in Python. This allows UCAT to interpret a Java code submission into an abstract syntax tree, and once it is represented by this data structure, the desired information can be extracted. The next tool used is MermaidJS[45] which uses a markdown language to programmatically create diagrams and charts. It supports creation of UML class diagrams, so once class information is extracted by the UCAT ANTLR implementation, it is a matter of translating it into the markdown language and handing it off to MermaidJS. MermaidJS then generates an image of the UML class definition which is inserted into the student feedback HTML file.

3.2.3 Feedback Pipeline

In order to provide meaningful feedback, UCAT improves upon its predecessor in two main ways. First, it extracts more information from the unit tests themselves, such as unit test name, assertion statement, and result, and lists them sorted by result type (passed, failed, skipped, etc.) in a feedback HTML file. Next, it includes a module for loading instructor feedback stored in a JSON file. This file allows instructors to write bits of feedback designated as “description” and “hint” for each unit test they create to assess the assignment. The

description can be used to inform the student of what the test is assessing and why, and the hint is intended to be presented when a unit test fails. This file is optional, and an instructor may choose to omit one or both pieces of feedback for a test. In addition to this, if relevant and possible, the feedback HTML file displays the class diagram of the student code submitted.

3.2.4 Data Output

A feature new to UCAT is the collection of student performance data. The UCAT predecessor collected snapshots of student code by storing a copy of the code at the time of assessment. UCAT does this as well as package the results of the unit test assessment into a JSON file. Data collected includes results data on a per-test basis, total performance results, and instructor settings like test weights and the feedback that was delivered. This data can be used to examine how changes to the autograder, the course, or other factors impact student assessment results.

3.2.5 Codio Interface and Environment

UCAT makes use of several features in the Codio environment to guarantee security in the execution of student code and final grade reporting. Security is enforced by the usage of a folder in the Codio Box called `secure` which requires root privileges to both see and access, so students are unable to interact with it in any way. This folder is where all instructor files, UCAT input files, UCAT itself, and collected performance data are stored. On top of this, Codio automatically sandboxes the student code within the Codio Box when it runs, preventing any harm or malicious activities from occurring. To report the final grade determined by the assessment, UCAT has a module that sends the results to the Codio grade book through the Codio Application Programming Interface (API). Even with these features, UCAT was designed to be modular in nature and could be adapted to an Ubuntu environment different than Codio if so desired. This would require a name change, however.

3.3 Architecture

UCAT makes use of multiple disparate software tools and environments in its implementation. The Ubuntu operating system is used as the glue linking them all, and the Python portion of UCAT invokes these various tools through Ubuntu system calls. This makes the tool very flexible as anything that can run in Ubuntu can theoretically be employed by UCAT; however, the downside is that the software stack for each of these tools must be configured, which is not always a simple task. The construction of UCAT can be generalized into four segments: the initialization Bash scripts, the core Python modules, the peripheral Python modules, and the external dependencies.

3.3.1 Bash Scripts

The Bash scripts are used by the instructor to configure and run the autograder. There are two: `SetEnvironment.sh` and `RunAutograder.sh`. When configuring the autograder for a given assignment, the instructor only needs to edit the environment variables exported in `SetEnvironment.sh`. These variables give UCAT information such as what language is being graded and the directories of all of the files it needs. This script then calls `RunAutograder.sh` which downloads the latest version of UCAT from a K-State Computer Science Department server and runs it, passing in the information configured by the previous shell script. When setting the Codio Box to use a custom grading procedure, `SetEnvironment.sh` will be called when a student clicks the button invoking the autograder.

3.3.2 Core Python Modules

The Python core of UCAT consists of several modules including the main module `CodioAutograder.py`, which has the `main()` function that runs when the autograder is first called by `RunAutograder.sh`. The other core modules include `CodioLib.py`, which handles communications to the Codio API, and `CodioLogger.py`, which handles UCAT event logs, the generation of student feedback files, and the creation of student performance

data. Additionally, there are two class files used to enforce data structures for passing information within UCAT. `TestResultsClass.py` stores information related to unit test results, and `ParsedLanguageClass.py` stores the class information extracted from student code for MermaidJS translation.

3.3.3 Peripheral Python Modules

The Peripheral Python Modules interface with the various tools used in UCAT's automatic assessment methods. These modules include the graded language handlers (currently `PythonHandler.py` and `JavaHandler.py`), the instructor feedback parser `FeedbackParser.py`, and the ANTLR files, which are language-specific, but will consist of Parser, Lexer, Listener, Visitor, and Visitor Implementation files for each language. This version of UCAT supports translation from Java to Mermaid, and the related files are `JavaParser.py`, `JavaLexer.py`, `JavaListener.py`, `JavaVisitor.py`, and `JavaVisitorImplementation.py`.

The language handlers conform to input-output specifications as expected by `CodioAutograder.main()` (see Appendix B), but aside from that their implementations can vary. For example, `PythonHandler.py` is short because it remains in the Python environment, calling upon PyUnit to assess the student code. The `JavaHandler.py` file, on the other hand, is more involved because it uses the aforementioned Ubuntu system calls to build and run a pre-configured Gradle[46] project that itself runs a Java wrapper. The instructor feedback parser is fairly simple; its functions read a JSON file and then match the unit test name associated with a feedback entry to the test names in the recorded test results. The MermaidJS translation signal chain is yet more complicated. ANTLR must be used outside of UCAT operation to generate parser, lexer, and visitor Python files for a desired language. Then, a visitor implementation file must be written by the UCAT developer to extract class information from a target file in that language. When these files are all included in the UCAT project, they can be used by the relevant language handler to create a class diagram of student code, a step which relies on the ANTLR runtime environment for Python. This process is also described in detail in Appendix B.

3.3.4 External Dependencies

On top of open-source Python packages (see the list in Appendix A), UCAT has a few dependencies external to its source code. This version supports two graded programming languages: Python and Java. UCAT is written in Python, and its environment is preinstalled with Ubuntu. The Java runtime environment must be installed separately along with Gradle, the tool used to build Java projects which is itself implemented in Groovy[47]. Furthermore, both MermaidJS and ANTLR must be installed in the Ubuntu environment. MermaidJS is implemented in JavaScript[48] and thus requires its runtime environment. For file generation, ANTLR runs in Java. Then, for the ANTLR-generated files used in UCAT to work, the Python runtime environment for ANTLR must also be installed. A summary of the technology stack and the required software versions is provided in Table 3.1, and installation instructions for each are provided in Appendix A.

Table 3.1: *UCAT External Dependencies and Versions*

Tool	Version
Ubuntu	22.04.1
Python	3.10
Java	11.0.17
Gradle	7.6
ANTLR	4.9.3
MermaidJS	9.3.0

3.3.5 Internal Systems Interactions

The relationships between the different tools used in this version of UCAT are represented in Figure 3.1. This figure groups the different tools by their language or environment. Arrows represent transitions between environments or tools and are labeled with the method of doing so, while line segments with solid bubbles represent important links within an environment. For example, the first Bash initialization script calls the second Bash script, which then directly calls the autograder transitioning from Bash to Python. The arrow from ANTLR to the Language Interpreter Files grouping is shown without fill to highlight the fact

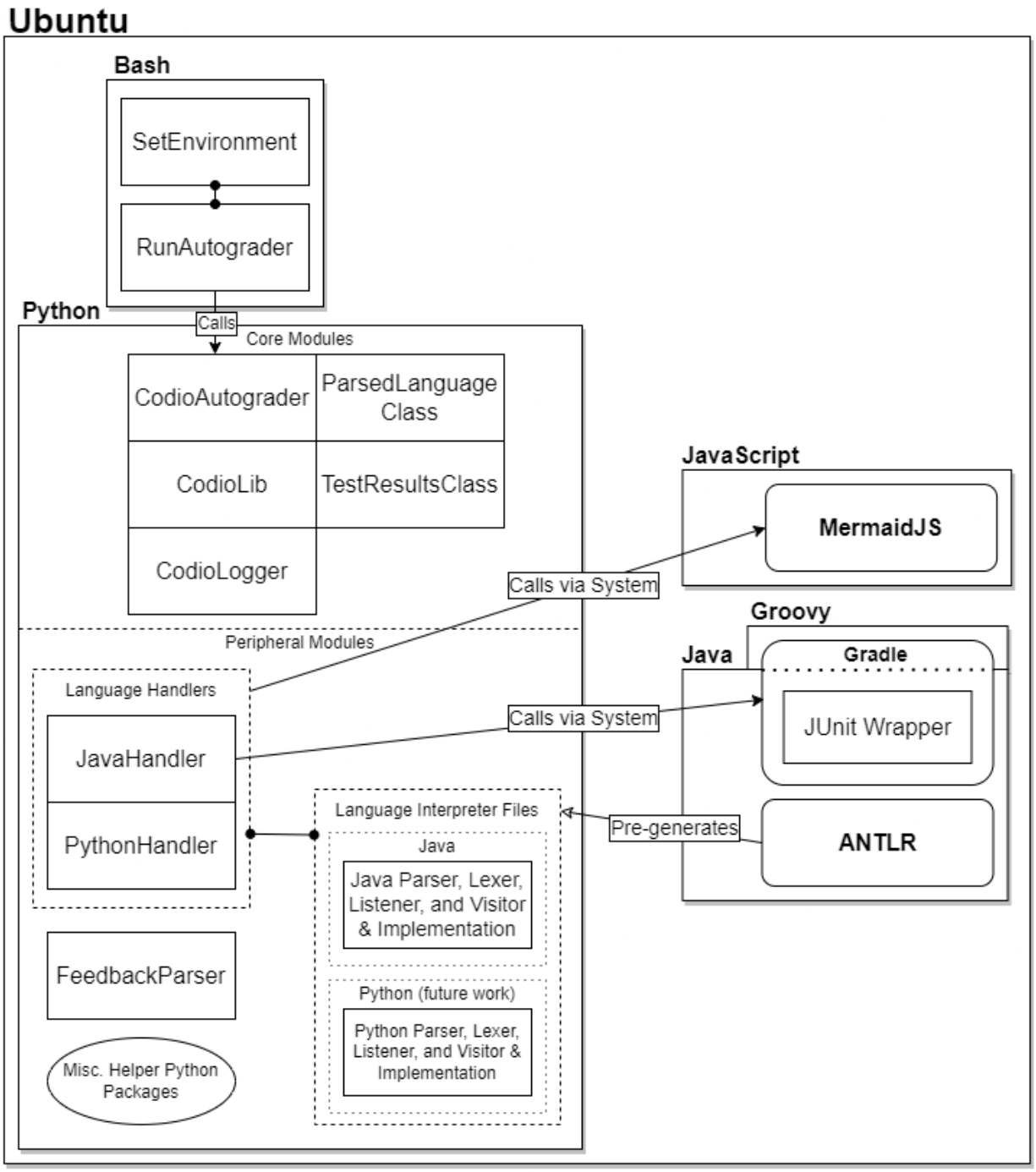

Figure 3.1: *UCAT Systems Interactions*

that its file generation process is done as part of the UCAT development process, not the automatic assessment process. Note that the ordered workflow of UCAT operations within environments is not represented in this diagram.

The origin and termination of each arrow is also important. Notice the arrow from the Language Handlers module grouping to MermaidJS. This is meant to indicate that calls to MermaidJS happen in a given Language Handler. On the other other hand, the Java Handler has an arrow starting at its border within the Language Handler grouping and ending at the Gradle border. This shows that the Java Handler specifically calls Gradle, the project builder that executes the Java unit testing wrapper. Furthermore, as mentioned, ANTLR is used to generate interpreter files for each language supported by UCAT, so its arrow terminates at the border of the Language Interpreter Files grouping. At time of writing, only Java to Mermaid translation has been implemented.

3.4 Operation

There are three main perspectives from which to use UCAT: the developer, the instructor, and the student. Figure 3.2 shows the main ways each stakeholder will interact with the system from the lens of the Codio Box.

The developer maintains an up-to-date version of UCAT in a K-State Computer Science server accessible from a Codio Box through the internet. The student interacts with Codio by using the web interface to develop and submit the programming homework assignment. When a student clicks the button in Codio labeled *Check It!*, UCAT is downloaded to the hidden folder `secure` by the `RunAutograder` shell script. In this way, students will always use the latest version. Hundreds of Codio Boxes can exist for a single class, so this method prevents needing to manually update them all when a new version of UCAT is released. The Instructor interacts with the system by configuring the Codio assignment and by providing files used by UCAT in the automatic assessment. To specify the assignment, Codio provides a markdown language and renderer called the Guide. On top of the Guide, the Instructor sets the variables in the `SetEnvironment` shell script, provides unit test files in the assignment language, and optionally provides per-unit-test feedback in a JSON file.

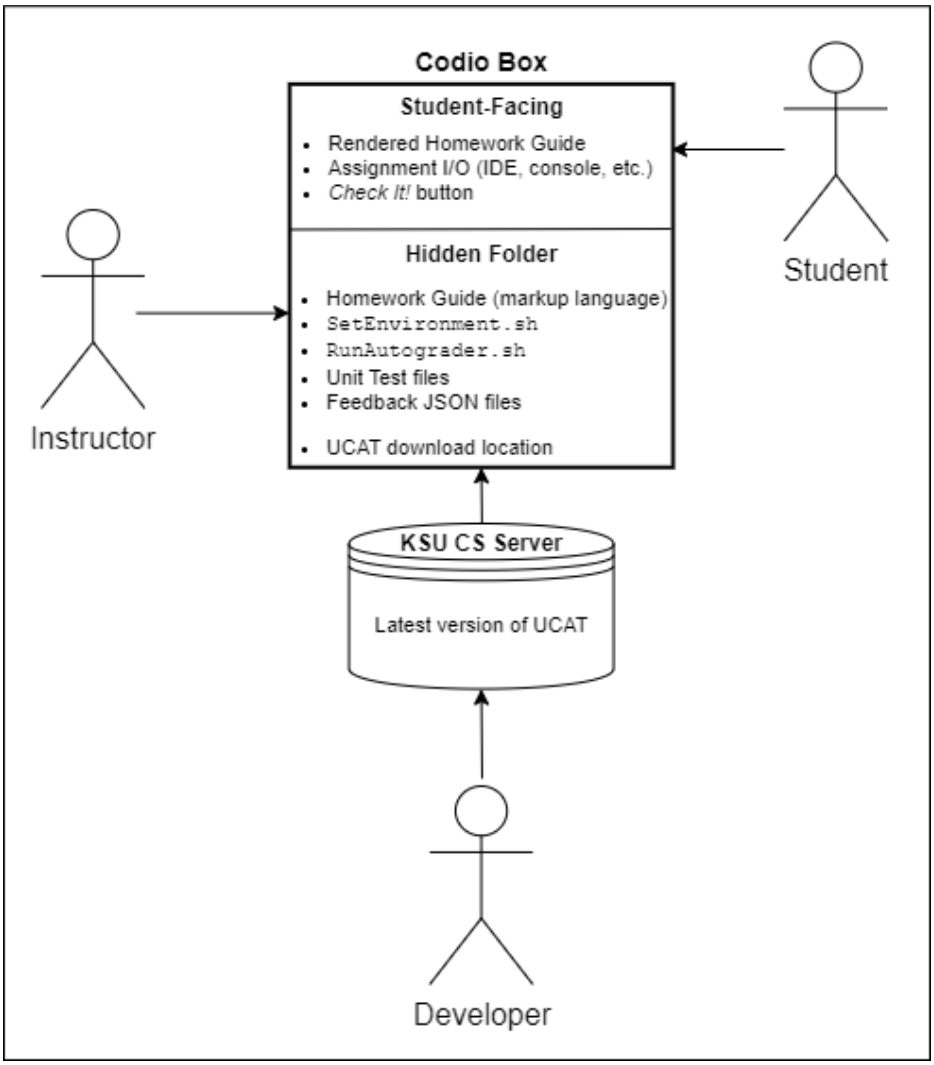

Figure 3.2: *UCAT Stakeholder Interactions*

The following sections will cover the need-to-know specifics for each perspective from the point of view of a new user. Because UCAT was designed for ease-of-use by both students and instructors, the developer perspective requires a great deal more background information.

3.4.1 As a Developer

This developer's guide includes information on the technology stack for UCAT, an overview of the source code, and information regarding extending UCAT functionality.

3.4.1.1 The Technology Stack

A developer new to UCAT should start with an Ubuntu machine, virtual or otherwise, at version 22.04.1. This version is what is used by the Codio Boxes. Python should come preinstalled with Ubuntu, but it should be verified that it is at least version 3.10. Before continuing, ensure the Ubuntu system is up to date. Specific commands to do this as well as install the following tools are provided in Appendix A.

Java and Gradle The Codio Boxes use Java 11. The latest version of Gradle at time of writing is 7.6, which supports Java versions through Java 19. Gradle is a build tool used to automate the execution of Java projects.

ANTLR There are multiple steps for installing and using ANTLR. ANTLR requires Java 11 or higher, so that must be installed first. It is important to install ANTLR manually instead of following ANTLR's quick-start guide that uses `antlr4-tools`. This is because not all ANTLR tools are needed, and the latest version of ANTLR is 4.12 which has a critical bug. The update from 4.9 to 4.10 introduced a format error in hard-coded serialized data written in the Python files created by ANTLR, so UCAT uses 4.9.3, the last version before this bug.

After ANTLR has been installed, parser, lexer, visitor, and listener files can be created using a context-free grammar paradigm specified by ANTLR. The ANTLR organization provides vetted grammars for many programming languages, and it is recommended to use these instead of attempt to create them from scratch. With this step complete, a visitor implementation file can be developed to traverse the abstract syntax tree of code parsed with these files. The ANTLR organization also provides obscure but syntactically correct code snippets that concisely test most all grammar rules for a language which is useful for testing a visitor implementation. For details on how to implement a visitor override file, extract information from an abstract syntax tree, and integrate it all into UCAT, see Appendix B and the existing Java implementation.

MermaidJS MermaidJS 9.3.0 (latest at time of writing) requires Node.js 18.13.0 and npm 8.19.3 or higher to run. If these are not yet installed on the Ubuntu system, the installation guide by Adeniran is recommended.[49] The Ubuntu command `sudo apt install` may not result in the proper versions of Node.js or npm installed, and this article provides an explanation and alternate means. MermaidJS depends on npm package Puppeteer[50] version 19.5.0, which itself has some Ubuntu library dependencies. MermaidJS will also be invoked via command line, and this interface must be installed separately.

3.4.1.2 Source Code Overview

The source code for UCAT is publicly available at the Kansas State University Advancing Learning and Teaching in Computer Science (ALT-CS) GitHub repository at `https://github.com/alt-cs-lab/`.

To demonstrate the full range and usage of the tool, the following sequence diagrams show UCAT assessment of a Java submission with instructor feedback provided and Mermaid translation enabled. The sequence is divided into five stages summarized in Table 3.2.

Table 3.2: *UCAT Example Sequence Diagram Summary*

Step	Phase	Figure	Section
1	Initialization	Fig. 3.3	Sec. 3.4.1.2
2	Unit Testing	Fig. 3.4	Sec. 3.4.1.2
3	Class Diagram Creation	Fig. 3.5	Sec. 3.4.1.2
4	Loading Instructor Feedback	Fig. 3.6	Sec. 3.4.1.2
5	Grade Reporting	Fig. 3.7	Sec. 3.4.1.2

UCAT Initialization Figure 3.3 details the invocation of UCAT and the initialization of its modules. A user first clicks the *Check It!* button in the Codio interface which is tied to the `SetEnvironment` script, configured by the instructor. This script sets environment variables and then calls `RunAutograder.sh` which does a few more operations before running UCAT's `main()` function. These two scripts are intentionally separated in order to simplify the instructor-facing interfaces to UCAT. Once UCAT is invoked, it parses command line

options taken from environment variables. This is where UCAT decides what features to run and what language handler to use. This phase also initializes the logger module and Codio API module. Something to note is that from this point on, the logger is passed around as an argument to many functions; this simplifies logging operations since a file may be opened just once at initialization and then written to by whatever the current context is (provided it has been passed the handle to the logger).

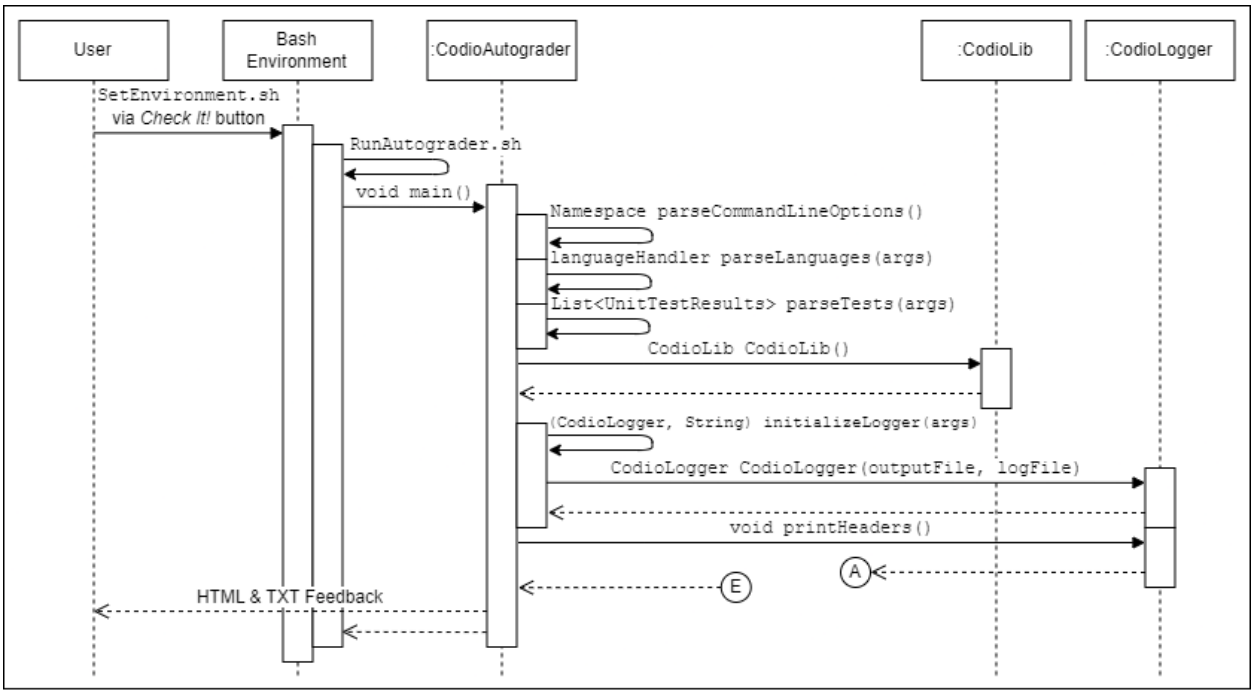

Figure 3.3: *Java Submission Sequence Diagram Part 1: Initialization*

A critical step in the initialization process is the creation of a list of `UnitTestResult` objects. This class definition stores the assessment information, including both unit test information and student results. An instructor is able to provide multiple files for test results and one `UnitTestResult` object is created for each. Among the fields of this class is a list of `UnitTestRecord` objects which store the specific unit test information. For the full definition of these classes, see the `TestResultsClass.py` file. After initialization and input parsing, the list of mostly unpopulated `UnitTestResult` objects is sent to the selected language handler. In this example, it is `JavaHandler.py`.

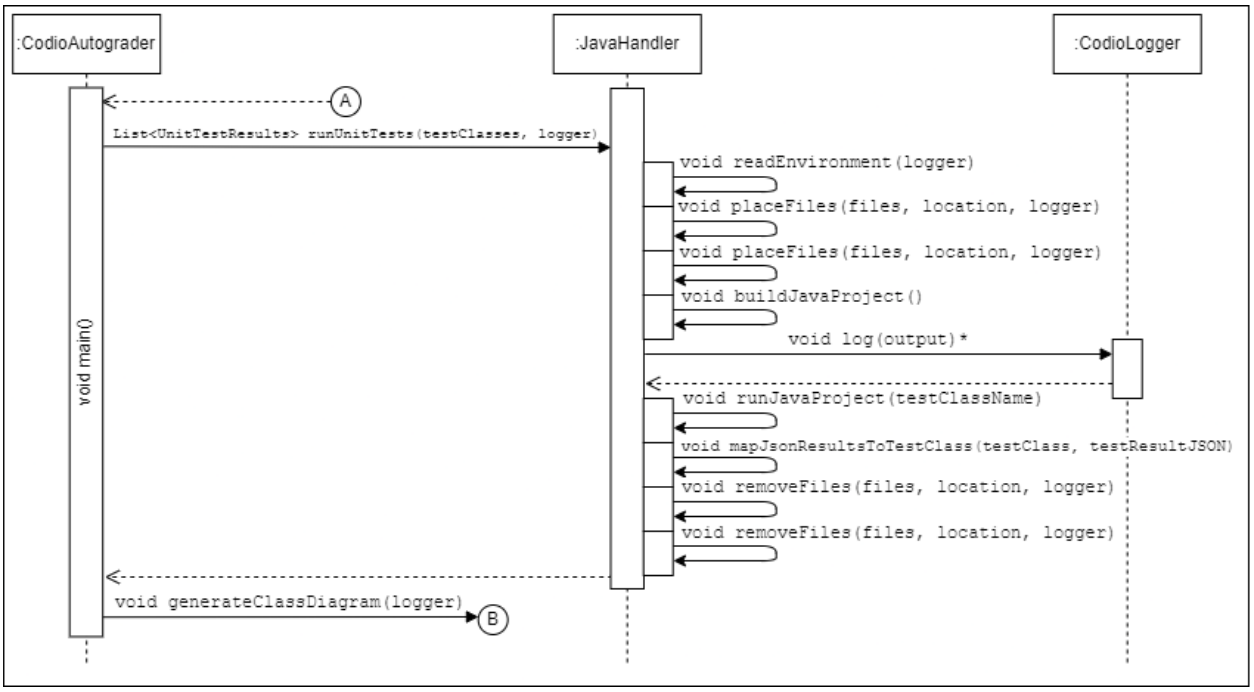

Figure 3.4: *Java Submission Sequence Diagram Part 2: Unit Testing*

Unit Testing One of the steps performed by `RunAutograder.sh` is the copying of both student and instructor files into the main directory. For Java, they must be copied again into the pre-configured Gradle project by the Java handler method `placeFiles()`, as shown in Figure 3.4. Once the files are copied to the proper locations, the Java handler uses Ubuntu system calls to build and then run the Gradle project. This is how the unit tests run against the student submission. The Java main function executing this also outputs the test results in a JSON-formatted string written to console output. The Python environment is able to read this information and maps it to the `UnitTestResult` object. Afterwards, the files are removed from the Gradle directories and the process is repeated for any more `UnitTestResult` objects representing instructor-provided files. Something to note in the diagram is a call to a function in the `CodioLogger` class shown as `void log(output)*`. The asterisk is meant to represent one or more unspecified logging functions that are left out of the sequence diagram for simplicity and conciseness. These logging functions populate the HTML student feedback and UCAT operations log TXT files. This representation is also seen in the a few of the following diagrams.

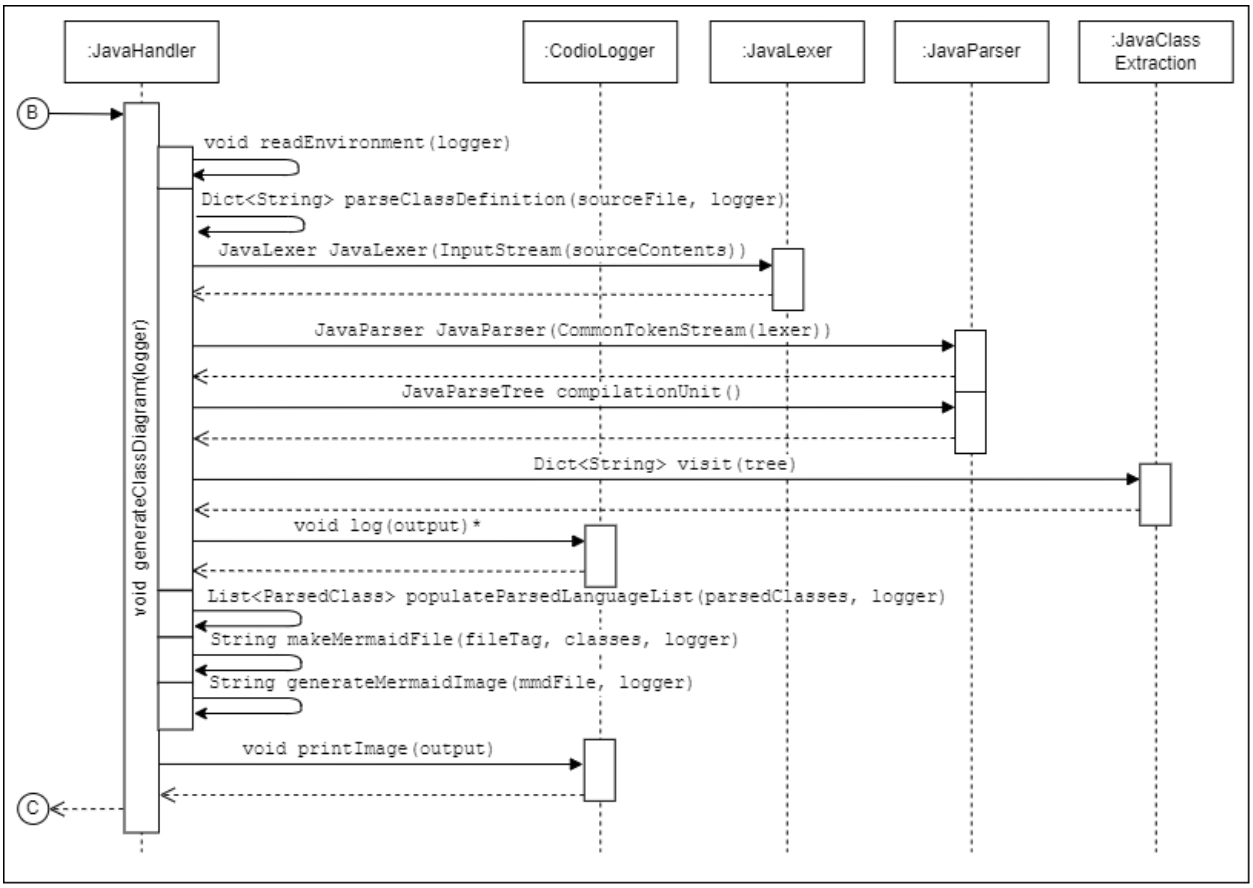

Figure 3.5: *Java Submission Sequence Diagram Part 3: Class Diagram Creation*

Class Diagram Creation Once the unit testing has successfully been completed, the language handler returns the list of now-populated `UnitTestResult` objects to the UCAT main function context. From here, it determines whether to attempt to create a class diagram image of the student code; in this example, it will return to `JavaHandler.py` to use the Java parsing functions created by ANTLR. This process is shown in Figure 3.5. For a more detailed discussion of this part of the sequence, see Appendix B. After the MermaidJS output image has been created, it gets inserted into the HTML feedback file by a `CodioLogger` function.

Loading Instructor Feedback With the class diagram created and added to the HTML file, the context again returns to the UCAT main function. The next step, shown in Figure 3.6, is to check for instructor feedback files and match their contents to the unit test data

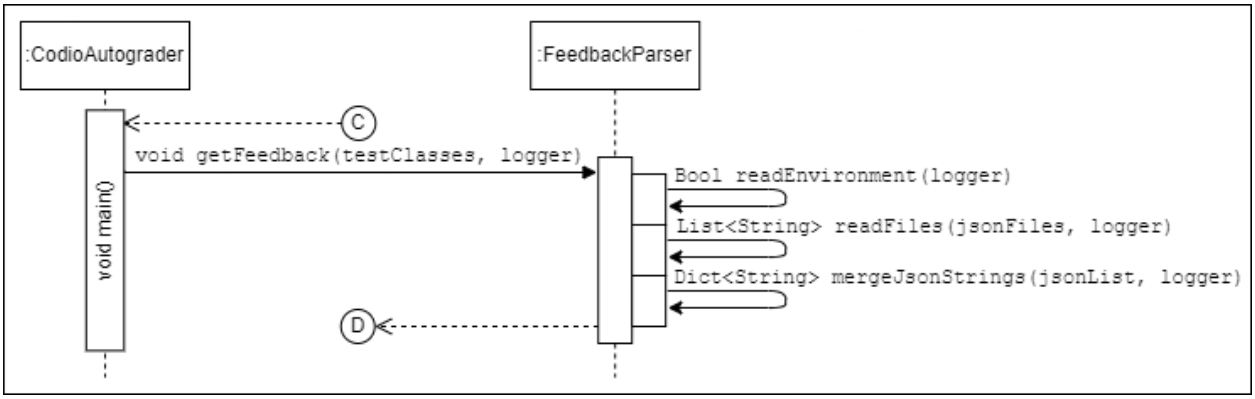

Figure 3.6: *Java Submission Sequence Diagram Part 4: Loading Instructor Feedback*

stored in the `UnitTestResult` object list. This file is in JSON format and is organized by unit test name. The test name in the JSON file must exactly match the name in the test file because this is how UCAT matches the instructor feedback to the proper `UnitTestResult` object. Each unit test entry has fields `description` and `hint`, each of which are optional. This file is designed to be flexible for instructors. An instructor may choose not to include a hint or description for some unit test, omit a full entry for a unit test, or omit the entire file.

Grade Reporting Final phase of UCAT is shown in Figure 3.7. Here, the context returns to the main function to finish operations. The submissions score is calculated based on weights provided by the instructor and the number of unit tests passed out of the total number of tests. The results are printed to log and student feedback files, the score is reported to Codio through the API, and then the log files are closed. From here, the sequence diagram ends, returning through symbol Circle-E. This leads back to Figure 3.3, the first in the process. UCAT terminates and returns to `RunAutograder.sh`, which ends by removing the copied-in student and instructor files to leave the environment clean for subsequent executions. The user will see the assessment results and feedback in the output HTML file and may peruse the output TXT file for more information recorded during UCAT execution.

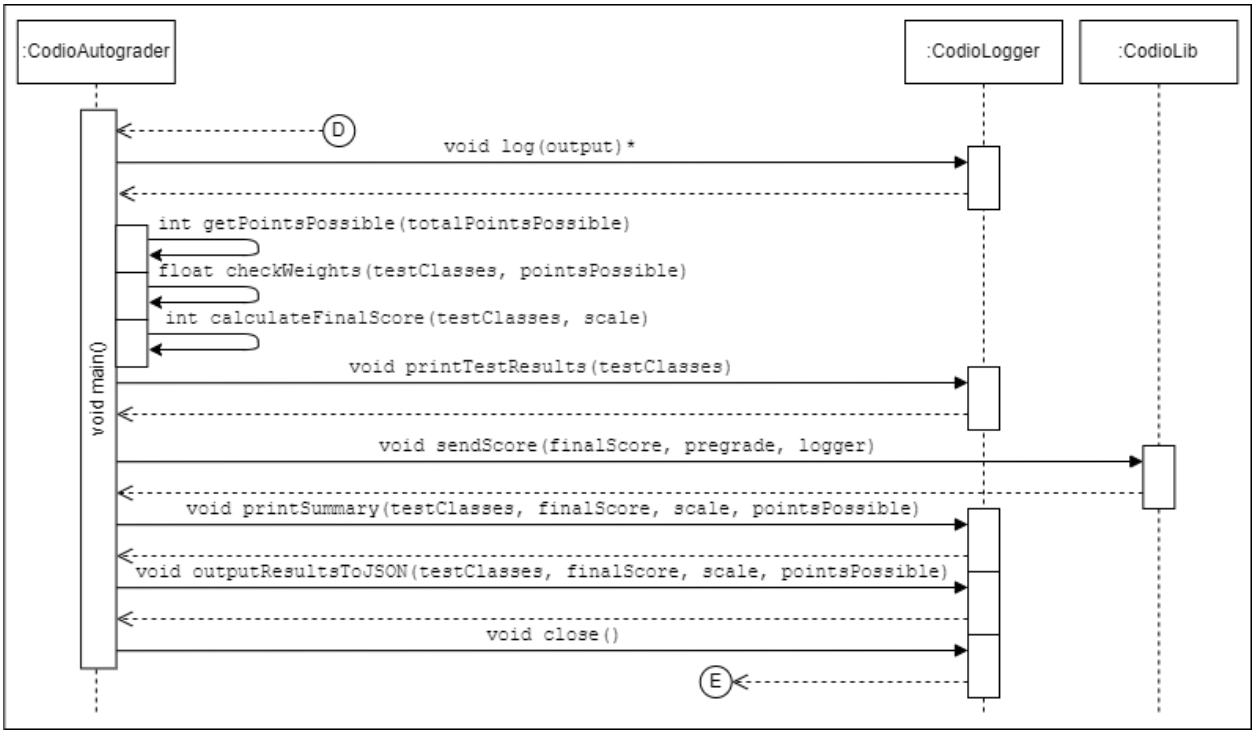

Figure 3.7: *Java Submission Sequence Diagram Part 5: Grade Calculation and Reporting*

3.4.1.3 Adding Support for New Languages

A strength of UCAT is that it has been generalized to support any language or tool that can run on or interface with an Ubuntu system. To add support for a new language, a few conventions and input-output requirements must be followed. Furthermore, if support for automatic creation of class diagrams is desired, there are separate and more complex steps to follow. As mentioned previously, the steps and requirements for each of these are detailed in Appendix B.

3.4.2 As an Instructor

Instructors should receive a pre-configured Codio Box from the departmental Codio administrator or developer before creating an assignment. The Codio Box will have everything set up for them to design the assignment; they will simply need to configure and provide the proper UCAT inputs (and make the Codio Guide). There are three parts to the Instructor UCAT inputs: configuring `SetEnvironment.sh`, creating unit tests, and populating the unit

test feedback JSON file.

3.4.2.1 Configuring `SetEnvironment.sh`

`SetEnvironment.sh` resides in the hidden `secure` folder in the Codio Box and contains a list of Bash `export` commands that set environment variables used by UCAT. These variables provide UCAT with file location information, grading weights, and more. The information that is configured by the instructor in `SetEnvironment.sh` is detailed in Table 3.3.

Table 3.3: *Instructor-Configured Environment Variables*

Variable	Description
`lang`	The graded programming language
`test_weights`	Unit test classes and weights, listed in pairs separated by spaces
`output_path`	Log file and location
`extract_path`	Location to store student code snapshot
`test_path`	Location of unit test files (assumed to all be in the same directory)
`test_files`	List of unit test files separated by spaces
`source_path`	Location of student source files (assumed to all be in the same directory)
`source_files`	List of student source files separated by spaces
`feedback_path`	Location of unit test feedback JSON file
`feedback_files`	List of unit test feedback JSON files separated by spaces
`flags`	Optional flags for configuring UCAT capabilities

Additionally, the Instructor may specify some optional UCAT features using flags. These optional flags are defined as a string separated by spaces in the `flags` variable and are summarized in Table 3.4. When the `-o` flag and weight are included, UCAT retrieves other assessment scores stored in the Codio environment for the assignment module. This allows flexibility to create multiple graded assignments in one Codio Box and have UCAT account for them all when calculating the final grade.

3.4.2.2 Providing Unit Tests

To score an assignment, UCAT needs at least one file of unit tests in the graded language with only one class definition per file. For security, these files should be stored in folder `secure`

Table 3.4: *Instructor-Configured Optional Flags for UCAT*

Flag	Description
`-p`	Pregrade; assess the submission but don't submit the score
`-s <int>`	Total points possible (defaults to 100)
`-o <int>`	Weight for other assessments in the module
`-m`	Generate UML class diagrams of student code via MermaidJS

which students cannot access. This directory may be structured however the instructor wishes, but the paths must be reflected in the configuration of `SetEnvironment.sh`.

3.4.2.3 Providing Feedback

Feedback for each unit test may be provided in the form of a JSON file. A template will be provided in the Codio box. Feedback is structured in the form of descriptions and hints. The description piece is displayed to the student in the HTML feedback file and is intended to provide information on what the unit test is assessing, while the hint gets displayed when a unit test fails. All parts of this are optional, meaning that an instructor may omit an entry for a unit test, may omit either the description or the hint, or may omit the file altogether.

3.4.3 As a Student

Students will have very little interaction with the inner workings of UCAT; they instead will work in the Codio web interface. Figure 3.8 shows an example of the Codio web interface for students. The far left panel displays the working directory of the Codio Box available to students; the middle panel is where students develop the code for the assignment. The right panel is the rendered homework Guide created by the instructor.

The button to call the autograder exists as part of the Guide. Figure 3.9 shows an example implementation with multiple assessment steps. When the Student clicks *Check It!*, the entire UCAT execution process is started. Upon successful completion, feedback files are created for them and the Guide is updated with console output from UCAT, shown in Figure 3.10.

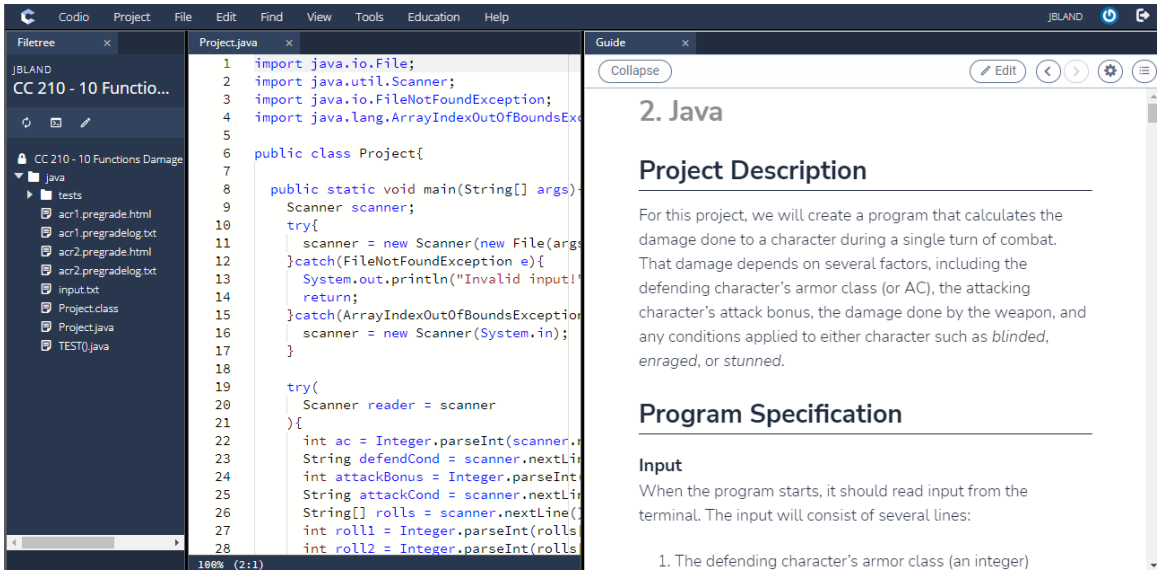

Figure 3.8: *Student Codio Interface*

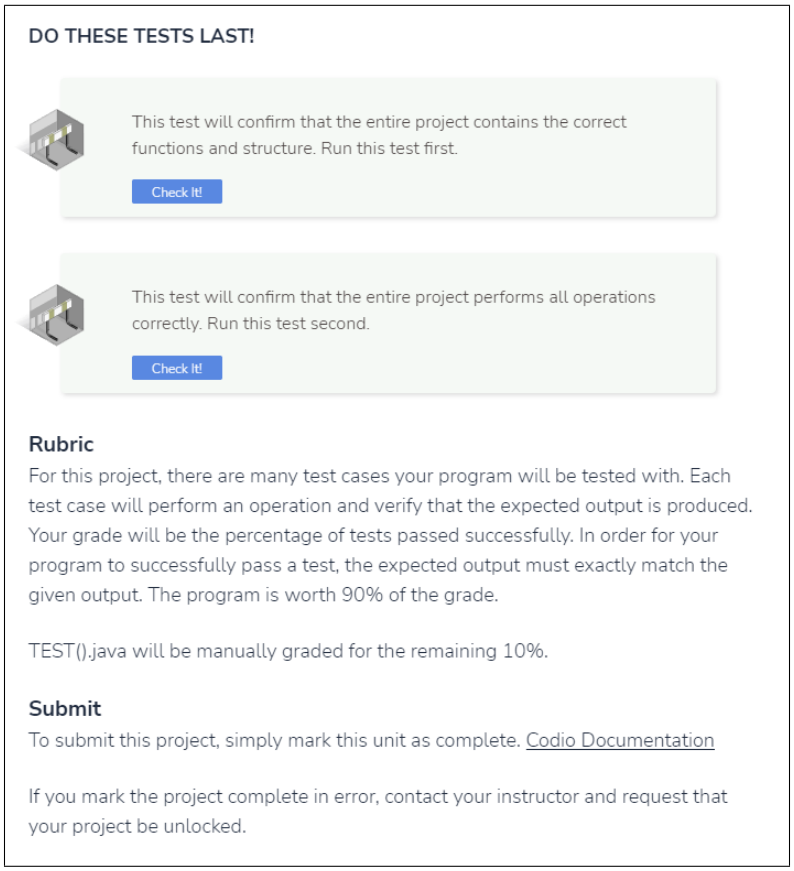

Figure 3.9: *Codio Check It! Button*

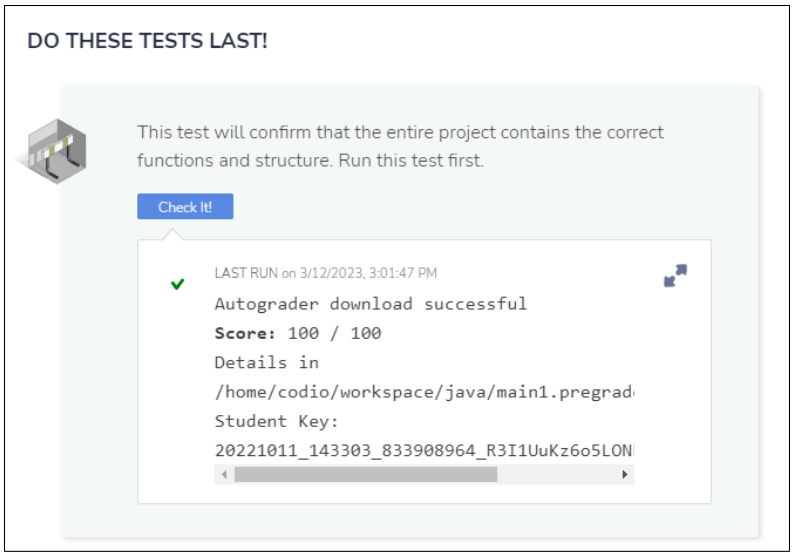

Figure 3.10: *Codio Check It! Results*

Student feedback is presented in an HTML page created by UCAT and viewable in the browser. This feedback incorporates unit test results, instructor guidance, final grades, and results of automatic static code analysis. Figure 3.11 shows the headers and the results of automatic static code analysis performed by UCAT. Figure 3.12 Shows the full feedback related to unit tests including the scoring statistics, test names, result assertions, and instructor descriptions and hints. The end of the HTML file reports a summary of the results including the final grade as shown in Figure 3.13.

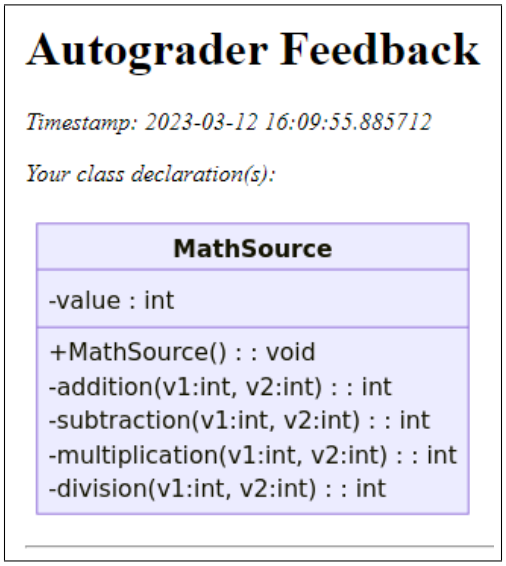

Figure 3.11: *UCAT Feedback: Static Code Analysis*

Test Class: MathTest1

Score: 50.0%

Weight: 100%

Statistics

- **Total tests**: 2
- **Passed**: 1 (50.0%)
- **Failed**: 1 (50.0%)
- **Errored**: 0 (0.0%)
- **Skipped**: 0 (0.0%)

Passed Tests

- `Test: test_subtraction`

  `Description: Checks that a subtraction operation is computed correctly.`

  `Result: Passed`

Failed Tests

- `Test: test_addition`

  `Description: Checks that an addition operation is computed correctly.`

  ```
  Result: AssertionError: Problem with addition
  Expected: <2>
       but: was <0>


  Hint: Check the operator used.
  ```

Figure 3.12: *UCAT Feedback: Unit Tests*

Summary

MathTest1: 50.0% * 100 = 50.0%

MathTest2: 100.0% * 100 = 100.0%

Scores will be scaled by 0.615 to make the total possible points 123.

Final Score: 92 out of 123 (74.8%)

Consult the log file for detailed output

Figure 3.13: *UCAT Feedback: Summary*

3.5 Limitations

UCAT implements all but one of the important autograder features identified in Section 2.3. The ability to limit or discourage overly many student submission attempts is not included in this version. However, given the flexibility of the tool's architecture, this would not be difficult for future developers. For example, a simple means of tracking submission attempts could be implemented using an Ubuntu environment variable.

UCAT is also designed as a feedback presentation engine rather than a feedback generator. This means that the quality and nature of the feedback depends on the instructor configuring the assignment. Furthermore, the feedback can't be tailored to the student, though that is a weakness of automated assessment in general. Additionally, with the assessment designed around unit tests, help can't be easily provided for syntax or runtime errors.

At time of writing there are a couple inefficiencies in the administration of the tool as well. Currently UCAT has to be manually moved to the download source location when there is a new update, but it is thought that this could be automated using code repository features. Though UCAT outputs assessment summary data files and records student code snapshots, this data is not currently aggregated into a database. The creation of such infrastructure and aggregation of existing data were considered out of scope for this project.

Finally, Python to Mermaid translation is not supported in this version. Java to Mermaid translation was successfully achieved and proved the concept, but the author must leave this functionality expansion to future developers, hoping the guides in the appendices will provide adequate assistance.

Chapter 4

Conclusions and Future Work

Implemented wisely, automated assessment tools can be an effective means of meeting the challenges of large course enrollments. Automated assessment has a long history in CS-Ed specifically, and the decades have provided insight into the best practices for autograder design. Autograders must secure student code from accidental or intentional system harm or academic dishonesty; they must assess the functionality of student code per instructor requirements; they must provide meaningful feedback to students; they should attempt some static code analysis; and finally, they should attempt to limit student reliance on the autograder.

This report presents an automated assessment tool for computer science courses titled UCAT: The Unified Codio Autograding Tool. UCAT interfaces with CS-Ed web service Codio, expanding its limited grading procedures. UCAT is designed around unit tests as a means of assessing student code, is feedback-focused with a pipeline for providing feedback directly from the instructor to the student, and executes automatic static code analysis that presents a class diagram of student code back to the student. Furthermore, it unifies the assessment efforts for different programming languages into one tool.

UCAT is modular in design for ease of future expansion. There are several obvious avenues for future work, with likely the most important being the creation of a means of limiting autograder submission attempts. This could be implemented as a grade penalty or

a simple limit on the number of attempts or time between them. More advanced courses could require students to provide their own unit test classes which would be easily integrated into UCAT.

Another aspect that could be improved in the future is the look and feel of the feedback HTML page. This version's output is bare-bones text on a plain white page; future developers could use CSS formatting to make the feedback more pleasant to view or even improve student learning. Gradle was chosen as a build automation tool for Java projects to create an interface between the Python and Java environments, but it does introduce much overhead when running Java code. In practice, this presents itself as a 30-60 second delay while the project builds. Future work could be done to minimize this wait time by streamlining Gradle operations to only those necessary for UCAT usage. Finally, UCAT could be expanded to handle more languages, depending on the needs of the institution. This version does not support Python to Mermaid translation for class diagram creation, but the instructions for doing so are provided in the appendices along with general language handler creation instructions.

In all, UCAT accomplishes its goals for improving the existing automated assessment tools used in Kansas State University Computer Science courses and is open for expansion and improvement by future developers.

Bibliography

[1] Kirsti M. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15:83–102, 2005. ISSN 17445175. doi: 10.1080/08993400500150747.

[2] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *ACM Journal of Educational Resources in Computing*, 5, 2005.

[3] M. Rifky I. Bariansyah, Satrio Adi Rukmono, and Riza Satria Perdana. Semantic approach for increasing test case coverage in automated grading of programming exercise. 2021. doi: 10.1109/ICoDSE53690.2021.9648439.

[4] Sylvia Alexander, Una O'Reilly, Pat Sweeney, and Gerry Mcallister. Utilizing automated assessment for large student cohorts. *Engineering education and research, 2001 : a chronicle of worldwide innovations*, 2002.

[5] Chris Wilcox. The role of automation in undergraduate computer science education. pages 90–95. Association for Computing Machinery, 2 2015. ISBN 9781450329668. doi: 10.1145/2676723.2677226.

[6] Stephan Krusche and Andreas Seitz. Artemis - an automatic assessment management system for interactive learning. volume 2018-January, pages 284–289. Association for Computing Machinery, Inc, 2 2018. ISBN 9781450351034. doi: 10.1145/3159450.3159602.

[7] Gerhard Hagerer, Laura Lahesoo, Miriam Anschutz, Stephan Krusche, and Georg Groh. An analysis of programming course evaluations before and after the introduction of

an autograder. Institute of Electrical and Electronics Engineers Inc., 2021. ISBN 9781728188836. doi: 10.1109/ITHET50392.2021.9759809.

[8] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. ACM, 2010. ISBN 9781450305204. doi: 10.1145/1930464.193048.

[9] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K Hollingsworth, Nelson Padua-Perez, and A V Williams Building. Experiences with marmoset: Designing and using an advanced submission and testing system for programming courses. 2006. doi: 10.1145/1140124.1140131.

[10] Sándor Király, Károly Nehéz, and Olivér Hornyák. Some aspects of grading java code submissions in moocs. *Research in Learning Technology*, 25, 2017. ISSN 21567077. doi: 10.25304/rlt.v25.1945.

[11] Elisa Baniassad, Lucas Zamprogno, Braxton Hall, and Reid Holmes. Stop the (autograder) insanity: Regression penalties to deter autograder overreliance. 2021. doi: 10.1145/3408877.3432430.

[12] Jack Hollingsworth. Automatic graders for programming classes. *Communications of the ACM*, 10 1960.

[13] Raymond Scott Pettit, John D Homer, Kayla Michelle Holcomb, Nevan Simone, and Susan A. Mengel. Are automated assessment tools helpful in programming courses? 2015.

[14] Codio. https://www.codio.com/, 2023. Create & deliver powerfully interactive computing courses proven to increase learner engagement & performance.

[15] Uml. https://www.uml.org/, 2023. Universal Modeling Language.

[16] Codegrade. https://www.codegrade.com, 2023. Deliver engaging feedback on code.

[17] Vocareum. https://www.vocareum.com/#classroom, 2023. Vocareum - cloud learning labs.

[18] Gradescope. https://www.gradescope.com, 2023. Online grading platform.

[19] Lab.computer. https://lab.computer, 2023. Put your computer laboratory on the browser.

[20] Web-cat. https://web-cat.org/, 2023. The Web-based Center for Automated Testing.

[21] Mike Joy, Nathan Griffiths, and Russell Boyatt. The boss online submission and assessment system. *ACM Journal on Educational Resources in Computing,*, 5, 9 2005.

[22] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education*, 19, 9 2018. ISSN 19466226. doi: 10.1145/3231711.

[23] Chung M. Tang, Yuen T. Yu, and Chung K. Poon. An automated system with a versatile test oracle for assessing student programs. *Computer Applications in Engineering Education*, 1 2022. ISSN 10990542. doi: 10.1002/cae.22577.

[24] Rémi Sharrock, Petra Bonfert-Taylor, Mathias Hiron, Michel Blockelet, Chris Miller, Mike Goudzwaard, and Ella Hamonic. Teaching c programming interactively at scale using taskgrader: An open-source autograder tool. 2019. doi: 10.1145/3330430.3333670.

[25] Claudia Ott, Anthony Robins, and Kerry Shephard. Translating principles of effective feedback for students into the cs1 context. *ACM Transactions on Computing Education*, 16, 1 2016. ISSN 19466226. doi: 10.1145/2737596.

[26] Peter Brusilovsky and Sergey Sosnovsky. Individualized exercises for self-assessment of programming knowledge: An evaluation of quizpack. *ACM Journal of Educational Resources in Computing*, 5, 2005. doi: 10.1145/1163405.1163411.

[27] Sadia Sharmin. Creativity in cs1: A literature review. *ACM Transactions on Computing Education*, 22, 6 2022. ISSN 19466226. doi: 10.1145/3459995.

[28] Valerie J. Shute. Focus on formative feedback. *Review of Educational Research*, 78: 153–189, 2008. ISSN 00346543. doi: 10.3102/0034654307313795.

[29] John Hattie and Helen Timperley. The power of feedback. *Review of Educational Research*, 77:81–112, 2007. ISSN 00346543. doi: 10.3102/003465430298487.

[30] Lucas Zamprogno, Reid Holmes, and Elisa Baniassad. Nudging student learning strategies using formative feedback in automatically graded assessments. pages 1–11. Association for Computing Machinery, Inc, 11 2020. ISBN 9781450381802. doi: 10.1145/3426431.3428654.

[31] Stephen H Edwards. Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance. 2003.

[32] Benjamin Clegg, Siobhán North, Phil McMinn, and Gordon Fraser. Simulating student mistakes to evaluate the fairness of automated grading. pages 121–125. Institute of Electrical and Electronics Engineers Inc., 5 2019. ISBN 9781728110004. doi: 10.1109/ICSE-SEET.2019.00021.

[33] Anton Dil and Joseph Osunde. Evaluation of a tool for java structural specification checking. pages 99–104. Association for Computing Machinery, 10 2018. ISBN 9781450365178. doi: 10.1145/3290511.3290528.

[34] Python. https://www.python.org/, 2023. Python is a programming language that lets you work quickly and integrate systems more effectively.

[35] Bash. https://www.gnu.org/software/bash/, 2023. Bash is the GNU Project's shell—the Bourne Again SHell.

[36] Ubuntu. https://ubuntu.com/, 2023. The open source software platform that runs everywhere from the smartphone, the tablet and the PC to the server and the cloud.

[37] Json. https://www.json.org/, 2023. JSON (JavaScript Object Notation) is a lightweight data-interchange format.

[38] Computational core. https://www.cs.ksu.edu/academics/computational-core/, 2023. The Computational Core is a set of computer programming courses designed to provide students of any major with the fundamental knowledge to utilize programming in a variety of situations.

[39] Docker. https://www.docker.com/, 2023. Use containers to Build, Share and Run your applications.

[40] Java. https://www.java.com/, 2023. Get Java for desktop applications.

[41] Hamcrest. https://hamcrest.org/, 2023. Matchers that can be combined to create flexible expressions of intent.

[42] Pyunit. https://wiki.python.org/moin/PyUnit, 2023. PyUnit is an easy way to create unit testing programs and UnitTests with Python.

[43] Junit. https://junit.org/, 2023. JUnit is a framework for writing repeatable tests.

[44] Antlr. https://www.antlr.org/, 2023. ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files.

[45] Mermaidjs. https://mermaid.js.org/, 2023. JavaScript based diagramming and charting tool that renders Markdown-inspired text definitions to create and modify diagrams dynamically.

[46] Gradle. https://gradle.org/, 2023. From mobile apps to microservices, from small startups to big enterprises, Gradle helps teams build, automate and deliver better software, faster.

[47] Groovy. https://groovy-lang.org/, 2023. A multi-faceted language for the Java platform.

[48] Javascript. https://developer.oracle.com/languages/javascript.html, 2023. Getting started with JavaScript.

[49] Adebola Adeniran. How to install node.js on ubuntu and update npm to the latest version. https://www.freecodecamp.org/news/how-to-install-node-js-on-ubuntu-and-update-npm-to-the-latest-version/, 2023.

[50] Puppeteer. https://www.npmjs.com/package/puppeteer, 2023. Puppeteer is a Node.js library which provides a high-level API to control Chrome/Chromium over the DevTools Protocol.

[51] How to install gradle on ubuntu 20.04. https://linuxize.com/post/how-to-install-gradle-on-ubuntu-20-04/, 2023.

Appendix A

Technology Stack Setup Instructions

To start, ensure the Ubuntu environment is up to date with this command:

```
sudo apt update && sudo apt upgrade -y
```

UCAT depends on a few helper packages in Python. Aside from `antlr4`, command `pip install <package_name>` should be sufficient to install these if they are not already. For details on installing `antlr4`, see Sec. A.2.

Table A.1: *UCAT Python Package Dependencies*

Package
os
json
sys
datetime
subprocess
shutil
antlr4
unittest
enum
argparse
requests
io
typing

Next, use the commands in the following sections to install the various tools used by UCAT.

A.1 Java and Gradle

To install Java, use the following command:

```
$ sudo apt install openjdk-11-jdk
```

To install Gradle, use these commands:

```
$ wget https://services.gradle.org/distributions/gradle-7.6.1-bin.zip -P /tmp
$ sudo unzip -d /opt/gradle /tmp/gradle-7.6.1-bin.zip
```

Successful installation may be verified with these commands, which will display version information for each:

```
$ java -version
$ gradle -v
```

For more information on installing Java and Gradle in Ubuntu, see the guide provided by website Linuxize.[51]

A.2 ANTLR

First, ANTLR must be installed in Ubuntu. To start, navigate to somewhere logical to store the library:

```
$ cd /usr/local/lib
```

Download ANTLR version 4.9.3:

```
$ curl -O https://www.antlr.org/download/antlr-4.9.3-complete.jar
```

Update the Ubuntu `CLASSPATH` to recognize it:

```
$ export CLASSPATH=".:/usr/local/lib/antlr-4.9.3-complete.jar:$CLASSPATH"
```

Create an alias to simplify calling the library:

```
$ alias antlr4='java -Xmx500M -cp "/usr/local/lib/antlr-4.9.3-complete.jar:
    $CLASSPATH" org.antlr.v4.Tool'
```

Changing the `CLASSPATH` and creating the alias will have to be done every time the system boots up, so those commands should probably be stored in a startup script. Test the installation by running the command `antlr4` with no arguments. It should bring up the version number and a list of options. Now the tool to generate the parser, lexer, and visitor for a grammar has been installed.

Next, install the ANTLR runtime environment for Python.

```
$ pip install antlr4-python3-runtime==4.9.3
```

Now the tool to use Python to parse some input against a grammar has been installed. To check that ANTLR has been installed correctly, create a simple grammar in a `.g4` file using examples in the official documentation.[44] The following command should create Python files for a lexer, parser, visitor, and listener based on the grammar defined in the `.g4` file:

```
$ antlr4 -Dlanguage=Python3 -visitor yourfile.g4
```

A.3 MermaidJS

MermaidJS 9.3.0 (latest at time of writing) requires Node.js 18.13.0 and npm 8.19.3 or higher to run, so these must be installed first. Setting up a Node.js project for the first time can be an ordeal; fortunately, the installation guide by Adeniran provides great instructions and is recommended.[49] The Ubuntu command `sudo apt install` alone may not result in the proper versions of Node.js or npm installed, and this article provides an explanation and alternate means.

Next, dependencies for npm package Puppeteer as well as Puppeteer itself (version 19.5.0) must be installed for MermaidJS to work.

```
$ sudo apt install libgtk-3-dev libnotify-dev libgconf-2-4 libnss3 libxss1
libasound2
$ npm i puppeteer
```

If problems are encountered with Puppeteer, try the toubleshooting guide at https://github.com/puppeteer/puppeteer/blob/main/docs/troubleshooting.md. Otherwise, MermaidJS may now be installed with npm.

```
$ npm i mermaid
```

Finally, the MermaidJS command line interface (CLI) must be installed:

```
$ npm install -g @mermaid-js/mermaid-cli
```

If these steps were successful, the following creation of a simple MermaidJS diagram should succeed. Make an `.mmd` file. Add this to its contents:

```
classDiagram
 direction RL
 class Student{-idCard :  IdCard}
```

Run this command, substituting your `.mmd` file name:

```
$ mmdc -i mermaidTest.mmd -o output.png
```

If everything was installed correctly, an image file called `output.png` should be created.

Appendix B

Extending UCAT Operation

UCAT was specifically written so that support for more graded languages may be added by future developers. This appendix details the conventions and requirements for extending assessment functionality as well as automatic class diagram creation.

B.1 Creating New Language Handlers

To add a new language handler, use the following process.

1. Add a python module with name `<Language>Handler.py` where `<Language>` is replaced with the name of the new language. This is the convention for the project. Don't forget to add `import <Language>Handler` to the top of `CodioAutograder.py`.

2. `CodioAutograder.py` contains a method called `parseLanguage()`. This method uses user inputs to know which graded language is being used and consequently which language handler to return. Thus, the `supportedLanguages` list and the if-statement tree in this method must both be extended to handle the new language.

3. The new language handler module must implement a `runUnitTests()` function that takes as arguments the unpopulated list of of `UnitTestResult` objects and a reference to the logger object. Internally, it can do whatever it needs to do in order to run the

unit tests, but it must populate that list with the results of the assessment and return it to the main function.

4. The new language handler must also implement a `generateClassDiagram()` function. This function takes as an argument the reference to the logger and does not return anything. If class diagram creation is supported for this language, its logic will be similar to existing implementations. If it is not supported, it is sufficient to print "Language class diagram creation not supported" or similar to the console and logger. For information on how to implement this, see Section B.2 and the UCAT Java implementation.

B.2 Adding Mermaid Translation Support for New Languages

The basic process for generating ANTLR files is outlined in Section A.2, but this guide will go into more detail. Assuming ANTLR 4.9.3 is properly installed, follow these steps to implement Mermaid translation for a new language. Side note: this section requires knowledge of programming language parsing and lexing, which is out of the scope of this document.

1. Check the official ANTLR GitHub repository for an ANTLR grammar definition for the language being implemented. The ANTLR organization has already done the work providing context-free grammars for many languages. These resources may be found at https://github.com/antlr/grammars-v4. They often define the lexers and parsers in separate `.g4` files, and both are needed for this process.

2. Once the `.g4` files are acquired, ANTLR may called to process them into Python files. The following commands use Java as an example.

For convenience, this command creates an alias for calling ANTLR. To avoid doing this every time the Ubuntu system boots, it could be stored in a startup script. Note that ANTLR is a Java application; the reference to Java in this command should not be confused with replaceable references to Java in subsequent commands.

```
$ alias antlr4=‘java -Xmx500M -cp "/usr/local/lib/antlr-4.9.3-complete.jar:
    $CLASSPATH" org.antlr.v4.Tool’
```

Next, use the alias to call ANTLR for processing the `.g4` files. The lexer must be processed first, which will create intermediate files used in the processing of the parser.

```
$ antlr4 -Dlanguage=Python3 -visitor JavaLexer.g4
$ antlr4 -Dlanguage=Python3 -visitor JavaParser.g4
```

This will create all of the files needed to interpret the language in a Python project (parser, lexer, visitor, and listener). However, there is one more quirk of ANTLR here. The parser and lexer Python files generated have an `import` statement for a package with an incorrect name; inside these files, `import` references to `typing.io` must be changed to `typing`. All of this so far can be automated with a script, and in fact it has been; the repository for UCAT contains a well-documented script doing this for any language. This script is called `GenTranslatorFiles.sh`.

3. Continuing with the Java example, files `JavaLexer.py`, `JavaParser.py`, `JavaParserListener.py`, and `JavaParserVisitor.py` should now be created. Copy these into whatever project directory is desired, and create a new file for the visitor implementation, possibly called `JavaVisitorImplementation.py`. This is where the logic to extract class information will be implemented using functions overriding those in `JavaParserVisitor.py`. Each function in `JavaParserVisitor.py` represents a possible node in the abstract syntax tree of a Java file as defined in the grammar. Not all node types will need to be accounted for when attempting to extract

class information. For specifics of Python class inheritance and function overrides as well as general strategy for extracting class information using this process, see UCAT file `JavaVisitorImplementation.py`. On top of providing vetted grammars, the ANTLR organization provides code snippets to test the parsing functions generated by the grammar files. These are concise but oblique blocks of code that are designed to test every single grammar rule for a language, so it is recommended to use them to test the visitor implementation to account for all syntax edge cases.

4. The next step is writing the correct syntax for using these ANTLR files in the language handler. As stated in Section B.1, each language handler must implement a method called `generateClassDiagram()`. This is where the interpretation and translation process is started. Implementation can vary per language handler, but the following Python syntax must be observed in order to correctly call the ANTLR files (continuing with Java as an example).

```
from JavaParser import *
from JavaLexer import *
from JavaVisitorImplementation import *

# Assume source is the handle to the Java file opened for reading
sourceContents = source.read()

# Create ANTLR lexer and parser objects
lexer = JavaLexer(InputStream(sourceContents))
parser = JavaParser(CommonTokenStream(lexer))

# Create the abstract syntax tree by calling the root node
```

```
# function as defined in the language-specific Antlr grammar.
# The root node for the Java grammar is called a compilationUnit
tree = parser.compilationUnit()

# Create the visitor object from the visitor implementation class
visitor = JavaVisitorImplementation.JavaClassExtraction()

# Traverse the tree using the visitor and return the class information
classInfo = visitor.visit(tree)
```

UCAT has a class for storing information about classes defined in student submissions. The `ParsedClass` class is defined in `ParsedLanguageClass.py` and should be used as the data structure for storing and passing the extracted class information.

5. From here, it is a matter of manually mapping the extracted class information stored in `classInfo` to the MermaidJS markdown syntax as defined at https://mermaid.js.org/syntax/classDiagram.html. This should be written to an `.mmd` file which can then used as input to MermaidJS for image creation (see Section A.3).