Alice Bibaud

Professor John Sturhman

Software Design and Documentation

28 July 2021

Final Exam

I hereby declare that all work in this "Exam" is original, and all sources used in this document are appropriately cited and sourced. See the list of references at the back of this document.

Signed:

[illegible]

Alice Bibaud
RPI CSCI '23

Table of Contents:

Question A

The FOSS

Submitty is a many-faceted grading application for computing classes and assignments. The repository is available here. Some of the main features are Secure Code Testing in many languages, including hidden tests; Customizable Automated Grading, which lends immediate feedback, error correcting, and grade summaries to students; Advanced Grading Tools for professors, including static analysis, JUnit, code coverage, and memory debuggers, all available through an instructor interface; and student upload of code, via drag-and-drop, zip upload, or version control system. The Version Control system could be its own feature, which allows students to correct mistakes through multiple submissions. Submitty also scales to multiple courses with thousands of students and supports multiple instructors and TAs per course.

Proposed Features

I am interested in creating two new features: an online compiler and a future grade calculator. These two features would be non-trivial to implement and would mesh well with already existing features.

The online compiler would be useful to students because there are certain constraints on programs run through Submitty that local machines do not catch. For example, a program that would normally run to completion on a student laptop might time out on Submitty, because Submitty has tighter time constraints than the student laptop. Thus, an online compiler would be a boon to students who have computers that can handle less efficient code.

An online compiler might also be of use to someone who does not have access to their own computer or to the memory debuggers required in classes like Data Structures and Operating Systems. Instead of requiring a laptop to work, a student could instead go to a computer lab or the library and test their code online. Thus, an online compiler would make accurate feedback on student code more accessible to those of us without consistent access to reliable computers.

The future grade calculator would be useful to students who want to ensure success in their computing classes by revealing the scores on assignments necessary for certain grade thresholds. For example, if a student is struggling in a class, they would be able to go to the future grade calculator, input the desired final grade they would like in said class, and see the minimum required score on every “gradable” (assignment or exam) they would have to achieve to reach their goal.

User Stories

Online Compiler:

- As a student, I want an online compiler in Submitty so that if my computer crashes, I can still test my code.
- As a student, I want an online compiler in Submitty so that if I can't install valgrind or doctor memory, I can still test for memory errors in my code.
- As a student, I want an online compiler in Submitty so that I know there are no important segmentation faults or time-out errors that I will lose points for in my code.
- As a professor, I want an online compiler in Submitty so my students are more aware of the edge cases and time and space efficiency I expect from their code.
- As a professor, I want an online compiler in Submitty so my students can better access Submitty when they have computer errors, thus reducing the amount of stressed emails in my inbox and late days used by students.

Future Grade Calculator:

- As a student, I want a future grade calculator in Submitty so that I can see what grades I need to get on my exams and assignments in order to achieve a B.
- As a student, I want a future grade calculator in Submitty so that I can know, by the drop deadline, whether or not I will be able to pass my class if I improve my performance.
- As a student, I want a future grade calculator in Submitty so that I can allocate time in my schedule to allocate time to classes I need to work harder in, and deallocate time where I don't need to work as hard.
- As a professor, I want a future grade calculator in Submitty so that fewer students fail my class.
- As a professor, I want a future grade calculator in Submitty so that I can tailor my curriculum to better suit the academic needs of my students.

User Scenarios

Personas:

Jesse, a freshman CS student at RPI, is taking Data Structures this semester. They have recently banded together with a few other CS students and have a friend group they can work on homeworks with. Jesse came from a little town in the midwest, where they excelled on the high school robotics team, and even had an unexpected, remote software engineering internship opportunity finding and fixing bugs in a new video game. As a result, Troy feels like an exciting new place that they are excited to live in for the next few years. They are doing well in the class. However, recently they got a 40% on a test whose average was 73%. As a result, Jesse needs to do very well on upcoming assignments and tests in order to pass the class.

Online Compiler Scenario:

Jesse needs to use Valgrind, a memory debugger, to make sure they have no memory leaks in their code. They pull up the Submitty website, and log in by typing their RCSID and password into the fields labeled “User ID” and “Password,” respectively. Now at the “My Courses” page, they see a blue box labeled “Summer 2021 CSCI 1200 Data Structures.” They click on it, and are brought to the “Compiler” page for that class. A list of assignments pops up; it includes current and completed assignments. Jesse selects “Homework 6,” which is due on July 14th. They select the file they want to test by clicking on the file browser, and press a green button near the lower left corner of the submission rectangle that reads “Run.” The amount of files left in the queue is written in red at the bottom of the screen as it is compiled and run, and when that is done, the output of the program is displayed in a pseudo terminal window, with all of the grading criteria and progress made on the assignment shown below.

Future Grade Calculator Scenario:

Jesse wants to know if they can raise their grade up to a B if they do well on Homework 6. They pull up the Submitty website, and log in by typing their RCSID and password into the fields labeled “User ID” and “Password,” respectively. Now at the “My Courses” page, they see a blue box labeled “Summer 2021 CSCI 1200 Data Structures.” They click on it, and are brought to the “Compiler” page for that class. Using the sidebar, they navigate from there to the “Rainbow Grades” page; it is formatted like a spreadsheet, with gradable categories and assignments labeling the rows. They scroll down to the row labeled “Homework 6,” with a blank field next to it. They type “1.0” into the blank field, which would represent getting a 100% on Homework 6. When they are done typing, the total “HOMEWORK” and “OVERALL” fields automatically recalculate and display Jesse’s projected total homework and final grades.

Responsibilities + Collaborators of Classes in Features

Compiler_Root Class:

Creates a large Compiler_Root class, which Autograder and Compiler both extend. Uses old methods from the previously un-extended Autograder class: get_testcases(), killall(), run_compilation(), generate_input(), and run_execution().

Compiler Class:

Modifies original run_execution(): different output is displayed to different fields. Modifies original generate_output(): output does not include total grade, and shows terminal output in the main display box rather than in a small section. Modifies original run_validation(): does not check for student users, does not check for plagiarism, does not verify course. Does not use archival methods, due to lack of version control here.

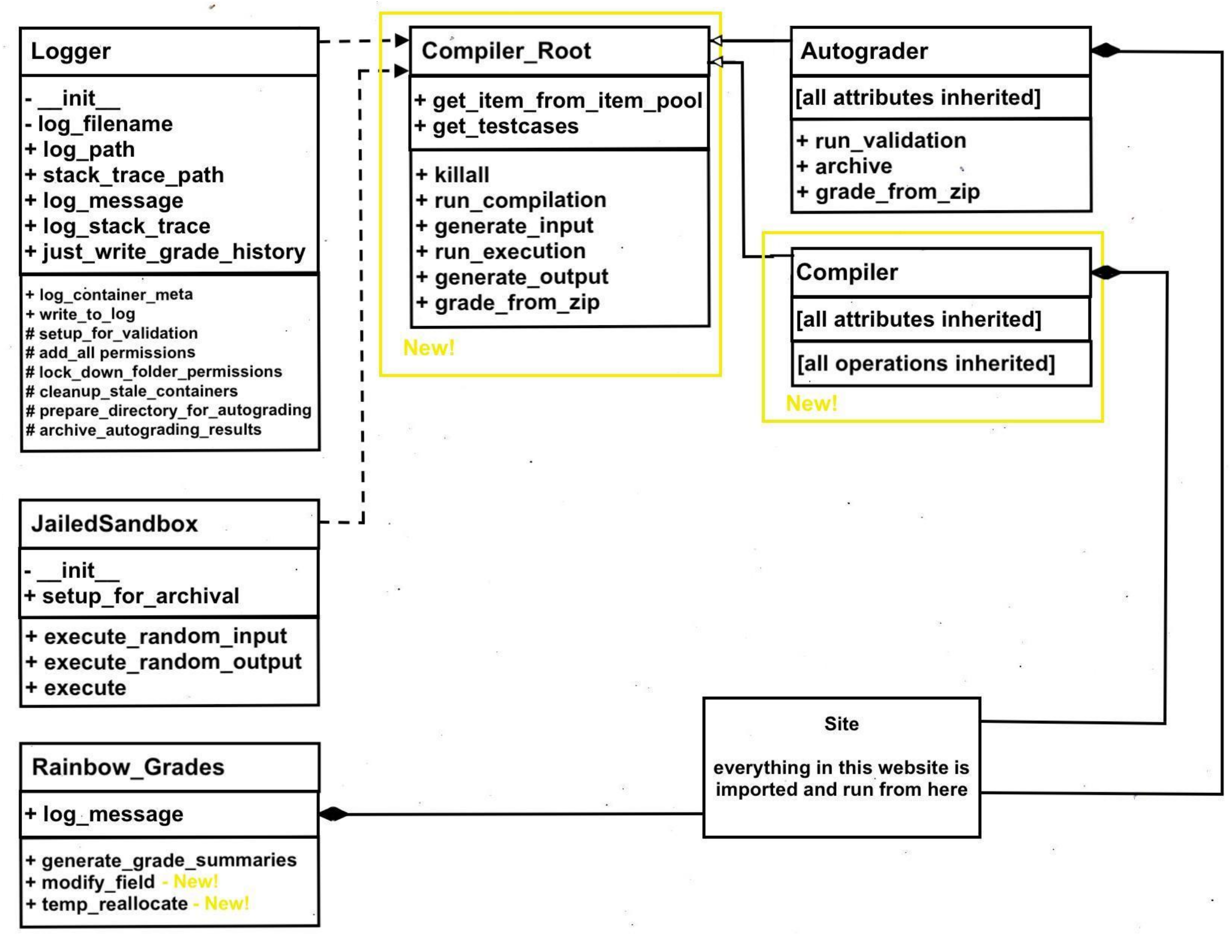
Imports files from autograding_utils as well as from execution_environments, thus taking the Logger class from autograding_utils, and the JailedSandbox class from execution_environments.

This class runs the same way as autograder - they are both imported into the site and used as parts of the whole website.

Future Grade Calculator:

This feature is entirely implemented within the Rainbow_Grades (private) class; the file it exists in is, unfortunately, not modular in its design. However, I will add two methods: mofidy_field(), which will modify single fields at a time, while recalculating total grades in each section of the course. I will also add a temp_recalculate() method, which will call the generate_grade_summaries, and show an uncolored potential grade, where the colored actual current grade stands in. When students navigate out of the file, temp_recalculate() will reset all fields to accurate colors and grades.

Class Diagram



Included are all the classes that the new classes directly rely on (dependencies), and it was, luckily, very easy to show what all of this looks like in context. The Site runs everything with containers, so all of the files are run all together as is. The Site does not have heavy specifications because there is not as much documentation on that topic, but the box included does a good job of showing that everything in the website runs from it.

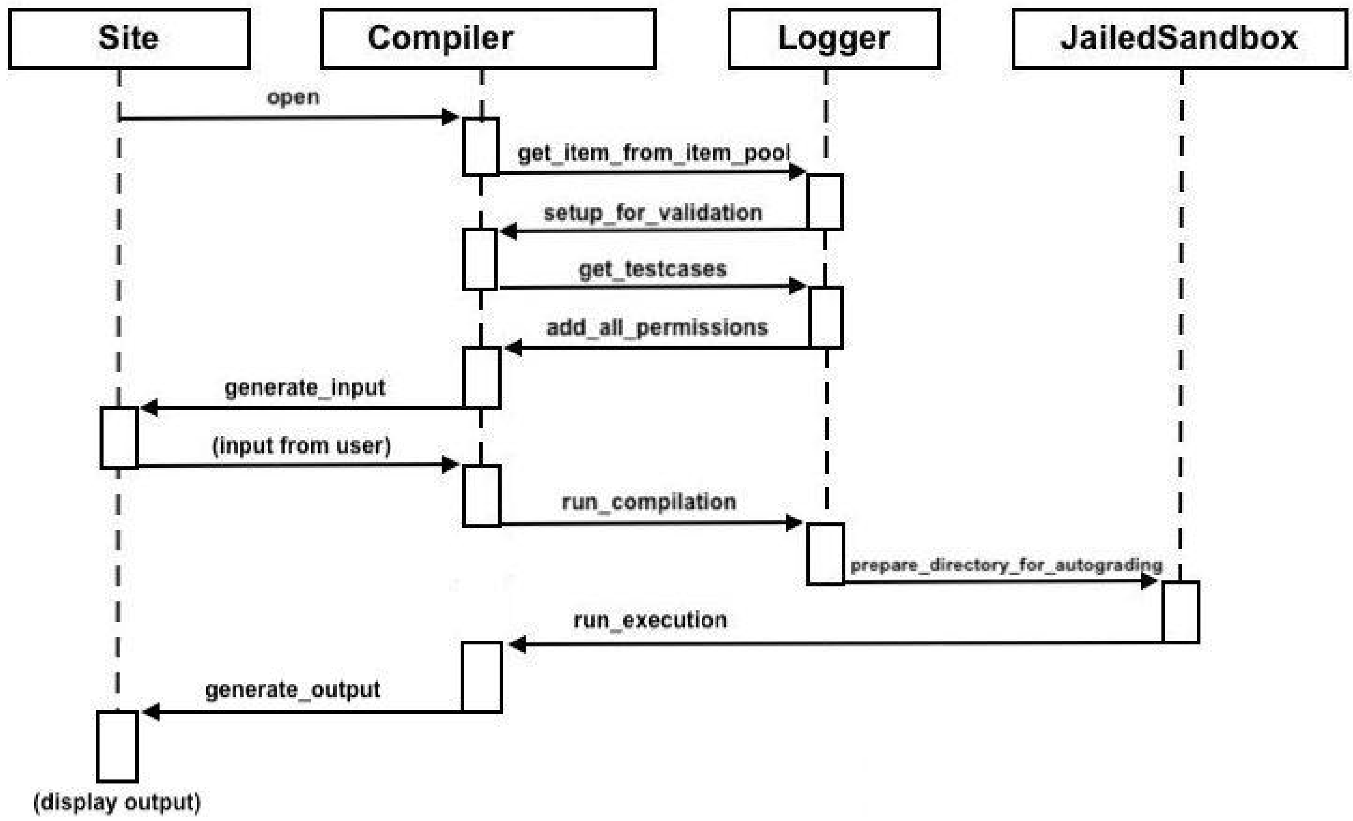
The Compiler_Root class depends on the Logger and JailedSandbox classes; Logger keeps track of version control and paths, which occasionally are output to the user, and the JailedSandbox is where the compiler is actually run. These classes have very strong preconditions. JailedSandbox requires the file from the user that is to be run, and information about the desired operating system and language with which a virtual machine will be built to deploy the user file. That information is passed to the program long before students submit their files. Logger has no preconditions; it looks at all current running containers, cleans up any unneeded processes, and prepares running containers for new user files. Compiler_Root has more preconditions: it inherits all preconditions from its dependencies (requires a virtual machine set up to spec, a student program from JailedSandbox, and clean containers from

Logger) as well as its instructor test cases that are to be run against the student file. Compiler_Root acts essentially as an operating system, in that it manages the processes passed to it. Because Compiler and Autograder extend almost every aspect of Compiler_Root, they both have the same preconditions as Compiler_Root itself.

Rainbow_Grades requires input from a class-wide and individual set of student grades, and, with the new feature, continually asks for input from the user. The set of student grades can be null or populated, and the user does not have to supply any input for it to work. This class outputs an HTML table populated with data on average and individual scores on assignments and tests any “gradable” made in Submitty, which is visible to users.

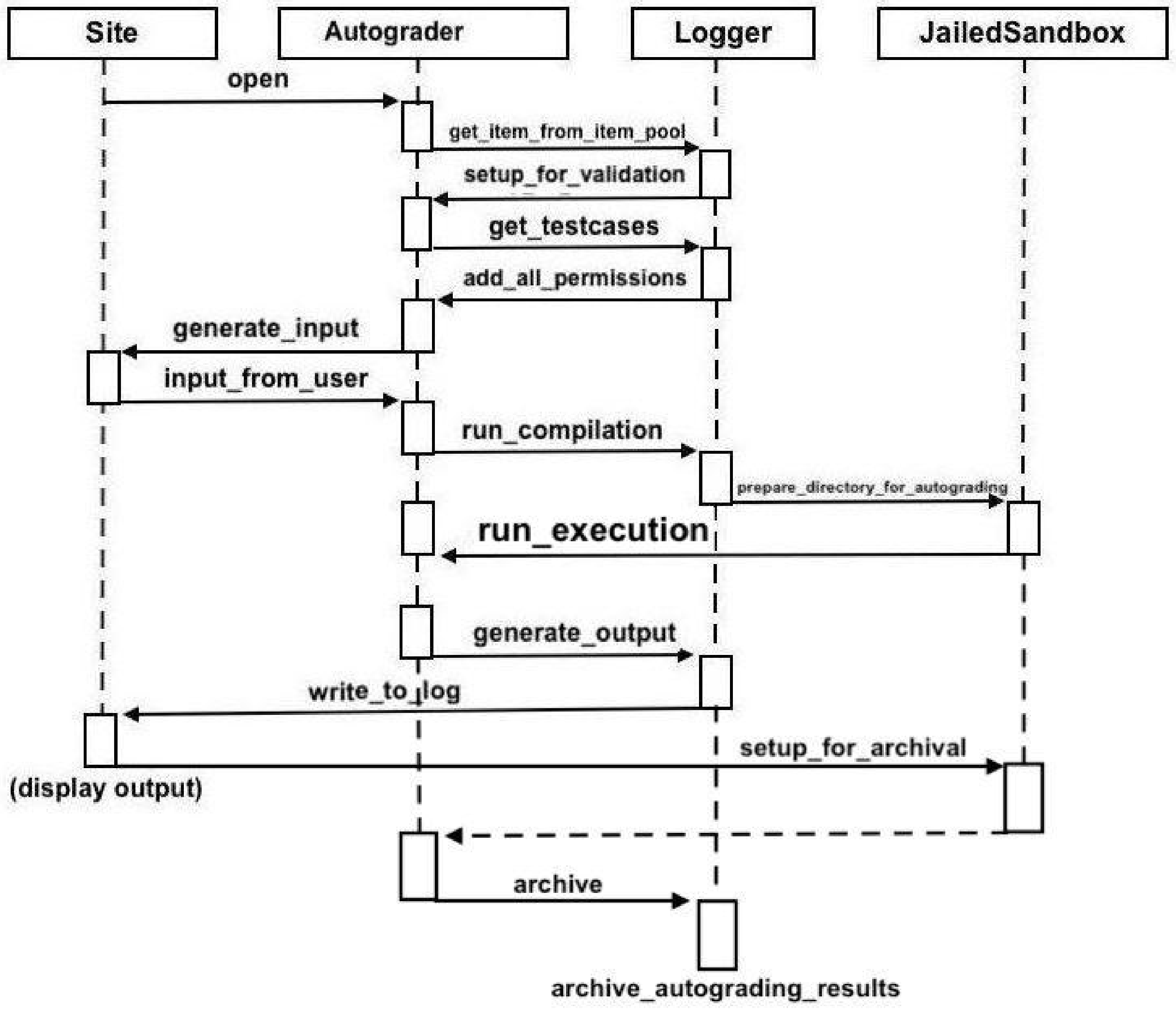
Sequence Diagram

Compiler:



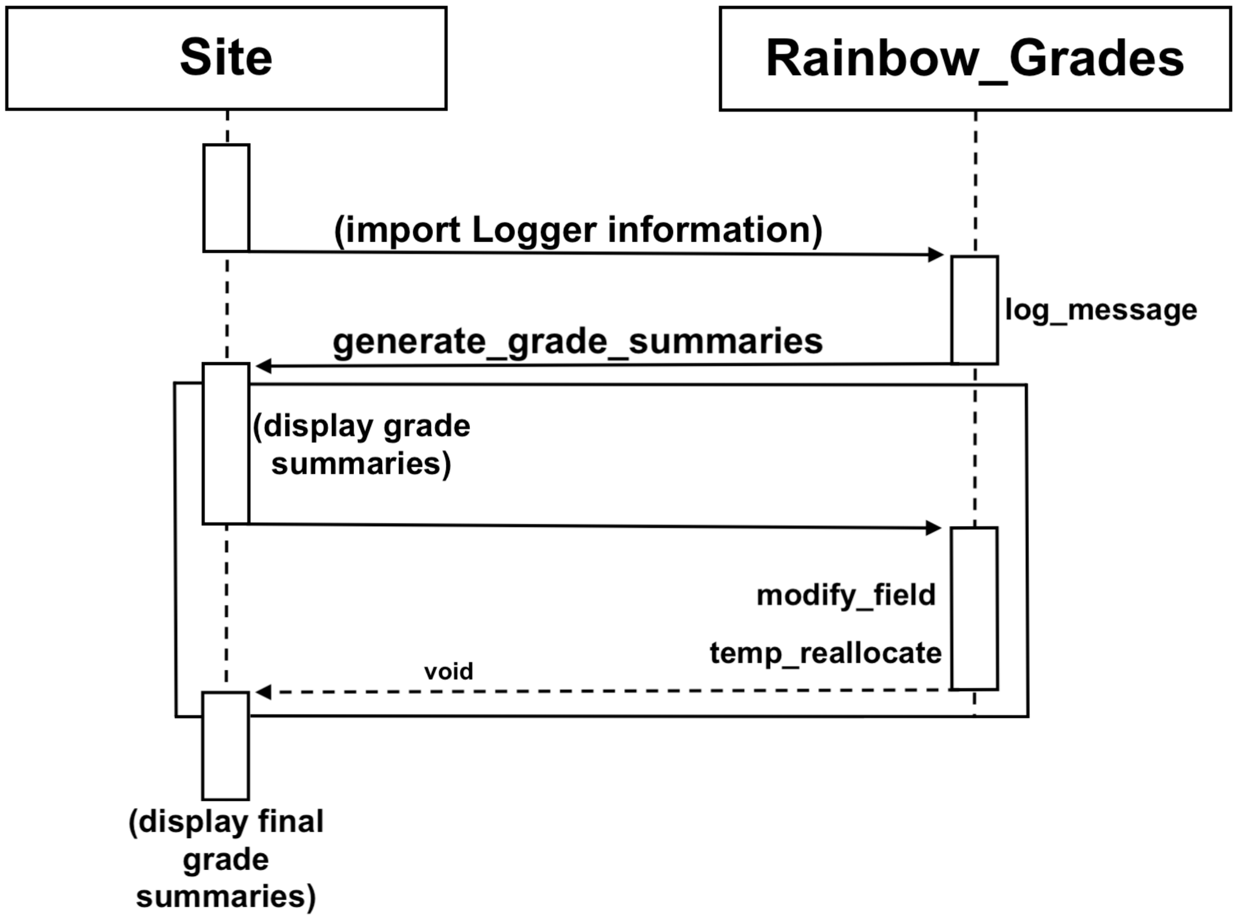
This sequence diagram shows how the user interacts with the site, and all of the behind-the-scenes action that would make compiling and running a program possible with the compiler. Notice how tightly intertwined the Logger and Compiler are; this is because Logger has links to the rest of the system. Including all of those connections would detract from the Compiler’s functionality and not clearly show the importance of JailedSanbox. The whole program runs from JailedBox, a virtual machine set up and then destroyed every time a program is to be run on Submitty. The Compiler and Logger must create and set up the sandbox before each use; you see that here.

Autograder:



The compiler and autograder run quite similarly. It is easy to see how, until output is displayed, the interactions between Compiler_Root, Logger, and JailedSandbox are the same as in the Compiler. However, the Autograder stores its results to be sent to the autograder, hence all the archival methods that are run after output are displayed to the site.

Rainbow_Grades:



Rainbow_Grades is far simpler than Compiler and Autograder. The whole potential grade calculator is built on a loop of displaying grade summaries and calculating what those grades would be. The feature's code terminates after a final grade summary is displayed. If there is no input from the user, the final grade summary is displayed anyway.

Additional Possible Feature

Another possible feature to add to Submitty would be another, multi-purpose compiler; instead of using existing gradeables as the means to compile student programs, a page would display a simple terminal window to the Submitty machine. That way, you could run any program with the Submitty protocols in place, and be able to see what differences between each computer would yield in output. This would also make the application even more accessible, because researchers, TA's, and professors could use Submitty's constrained resources as a way to test their programs more efficiently, to see what their files would do with less room for error, especially in the way of time.

This would be even more trivial than the current proposed compiler to implement because you would be able to use a C program that fed in a constant input stream that would then be read in and executed as terminal input; there are commands built into the C library for this express purpose.

SOLID Principles

Single Responsibility Principle (SRP):

The SRP means that each class should only have one responsibility. If there is more than one responsibility per class, this results in coupling, rigidity, and fragility in the code base. The compiler class obeys the SRP because of the way I reconstructed it: I created a large abstract class (Compiler_Root) to get rid of unnecessary repetition and minimize coupling between the autograder and the new compiler. The future grade calculator obeys the SRP in that Rainbow_Grades continues to have just one responsibility: calculate the grades the student has in the class. There are no new classes created, so there are no new dependencies; every feature acts internally and, therefore, does not create any new issues outside the class.

Open Closed Principle (OCP):

The OCP means that each class should be closed for modification but open for extension. This requires that when the original module is used, it can only be added to; this is achieved through abstraction. The compiler class does this by design: the Comipler_Root class I created to hold the common attributes and operations of the compiler and autograder is extended upon by its "child" classes whenever it is used, a good example of this abstraction. Rainbow_Grades subscribes to the OCP in that the system it was built to support already had a modular functionality; each piece of the whole gets sent out to the Site, and the Site deploys all of the files within it.

Liskov Substitution Principle (LSP):

The LSP is based upon the idea that subclasses should only have stronger specifications than their parent classes. Subclasses should be substitutable for their base classes, and derived methods should have fewer requirements and stronger post-conditions than their inherited methods. None of the features I've created have any non-abstract inheritance, so there is no way to disobey the LSP. However, if the compiler and autograder classes did indeed extend the Compiler_Root class, the methods used by both would need to be usable if each instance of the child class was replaced by their parent class.

Interface Segregation Principle (ISP):

The ISP posits that clients should not be dependent on classes they do not use. This principle highlights the need for modular code and depends itself on the open-closed principle; if each part does not work independently from each other, then your code base will become so fragile it will be unusable. Once again, the Rainbow_Grades class is not implicated by the ISP, because Rainbow_Grades itself does not have any dependencies. However, the Compiler_Root, compiler and autograder classes show the ISP in that I have eliminated all previous unnecessary class dependencies the autograder once relied on. Because Compiler_Root is an abstract class, it can pass only what is needed to its children, and no waste of resources or durability will be incurred.

Dependency Inversion Principle (DIP):

The DIP is a double-pronged idea. The first: high level ideas and low level ideas should not be mutually dependent, but connected through abstractions. The second: abstractions should not depend on details, they should depend on abstractions. Once again, the Rainbow_Grades class is not implicated by the DIP because Rainbow_Grades itself does not have any dependencies. However, the Compiler_Root, compiler, and autograder classes exemplify the DIP in that the potential dependencies of the lower level compiler on the higher-level autograder were eliminated when their connection, Compiler_Root, was abstracted.

Ease of Change

Compiler:

How many classes would I have to add for this new feature? I only had to add two; one to interface with pre-existing code and prevent duplication, and one that would do the work of the main feature.

How many classes would I have to change to add new types? To add new types to the Compiler, you would only need to add as many classes as new types; they would all extend the Compiler_Root class, so all the new types would be integrated modularly into the system.

How many classes would I have to change to support extensions? You should not need to change any classes to support extensions. The Submitty code base adheres to the Open-Closed Principle (OCP), so every feature added is open for extension and closed for modification.

How many classes would I have to change to support new properties? You should not need to change any classes to support new properties. The Submitty code base adheres to the Open-Closed Principle (OCP), so every feature added is open for extension and closed for modification.

Future Grade Calculator:

How many classes would I have to add for this new feature? No additional classes need be added to this feature. The infrastructure for it is already built into a similar, existing feature, and the development would be more efficient if we keep the environment streamlined, so no more classes are needed.

How many classes would I have to change to add new types? To add new types to rainbow grades, you would not have to add new classes, unless they were especially large; many features are easily integrated into this system without an especially large amount of code.

How many classes would I have to change to support extensions? After this new feature is added, no classes need be changed to support extensions. The Submitty code base adheres to the Open-Closed Principle (OCP), so every feature added is open for extension and closed for modification.

How many classes would I have to change to support new properties? After this new feature is added, no classes need be changed to support new properties. The Submitty code base adheres to the Open-Closed Principle (OCP), so every feature added is open for extension and closed for modification.

Submitty: Question B

The Methodology

The most realistic plan for developing Submitty is with Crystal. Crystal is unique among other Agile processes in that it is scalable to any size, allows team members autonomy in the development process and is completely contingent upon good communication practices. The methodology was developed by Alistair Cockburn in 1991 under the direction of IBM; he realized that teams perform differently depending on team size and project priority, hence why Crystal so fluidly deals with scalability.

Core Concepts and Values

Frequent Delivery:

Working code must regularly be released. This ensures that the product is valuable, and facilitates the finding and fixing of bugs.

Reflective Improvement:

Crystal hinges on frequent, team-wide discussions about the successes and failures within the development process. These talks ensure future success by focusing on what works.

Osmotic Communication:

Osmotic Communication is the free-flow of ideas of people in a shared space. This ultimately reduces the need for overhead involvement since the robust communication enabled by physical proximity removes the need for super involved managers.

Personal Safety:

There are no wrong answers in Crystal; Personal Safety ensures that each team member has a place to speak their mind.

Focus on Work:

Crystal depends on team members having clear work assignments and allowing developers long periods of work without interruption. This kind of focus hastens completion of the final product and prevents wasted work.

Access to Experts / Users:

Access to Experts / Users speeds up development by enabling communication with actual users of the product.

Technical Tooling:

Development teams should have access to the latest and greatest versioning software so bugs can be caught quickly.

Frequent Integration (only applies to large teams):

In large projects, it is very important for branches to be merged often, so bugs in branches are caught and fixed early in the development process.

Through these guiding concepts, it is clear that communication and good teamwork are the main values of the Crystal methodology.

Roles

Executive Sponsor:

Is in charge of the business side of things; they manage the money and make "company-wide" decisions.

Ambassador User:

Does the final testing of the project. The Ambassador User is meant to act like a user, so their input is used in other parts of development as well. They also have knowledge of the whole system as it is meant to be used.

Lead Designer:

The "head developer" in a classical setting: they are in charge of all technical work and make sure the rest of the team is on track.

Designer-Programmer and Development Team:

They work alongside the Lead Designer to make sure all new ideas are feasible within the code base.

Coordinator:

The “project manager” in a classical setting: they keep sponsors informed of all progress made so far, take notes at meetings to ensure promises made there are met, and keep the Project Sponsor up to date about the development process.

Business Expert:

The Business Expert knows how the business runs, prioritizes development, and is the main point of consultation for the rest of the development team.

Tester / Technical Writer:

The tester is either a rotating or temporary role. The tester tests the software, reports bugs, and attempts to fix them.

Artifacts and Ceremonies

Artifacts in Crystal are more loosely defined than in Scrum; different team sizes will have different needs for artifacts. The amount of verbal communication between developers required in this Agile methodology discourages documentation and other more formal artifacts. However, a dictation of a conversation from a reflection workshop, or a list of tasks to be performed by a specific team member during an Episode could be construed as such an artifact. More formally defined are certain deliverables and ceremonies common to all team sizes in Crystal:

Episodes:

Like Scrum, these are small increments of time and work that have focused goals. The work of a project is broken down in Episodes. Unlike Scrum, all episodes do not need to be integrated immediately into the final product. The robust communication characteristic of Crystal enables this flexibility.

Integration:

Because not all episodes are immediately integrated into the project, there is a need for deliberate integrations of episodes into the final product. These happen every two or three episodes.

Reflection Workshops:

These happen after each Episode; the whole team gathers together and talks about their successes and failures. The Coordinator takes notes at these and enforces the good ideas within groups later.

Deliverables

Everyone:

- Team Structure and Conventions
- Reflection Workshop Results

Sponsor:

- Mission Statement
- Tradeoff Priorities

Coordinator:

- Project Map
- Release Plan
- Project Status
- Risk List
- Iteration Plan & Status
- Viewing Schedule

Business Expert + Ambassador User:

- Actor-Goal List
- Use Cases & Requirements File
- User Role Model

Lead Designer:

- Architecture Description

Designer-Programmer and Development Team:

- Screen Drafts
- Common Domain Model
- Design Sketches + Notes
- Source Code
- Migration Code

- Tests
- Packaged System

Tester / Technical Writer:

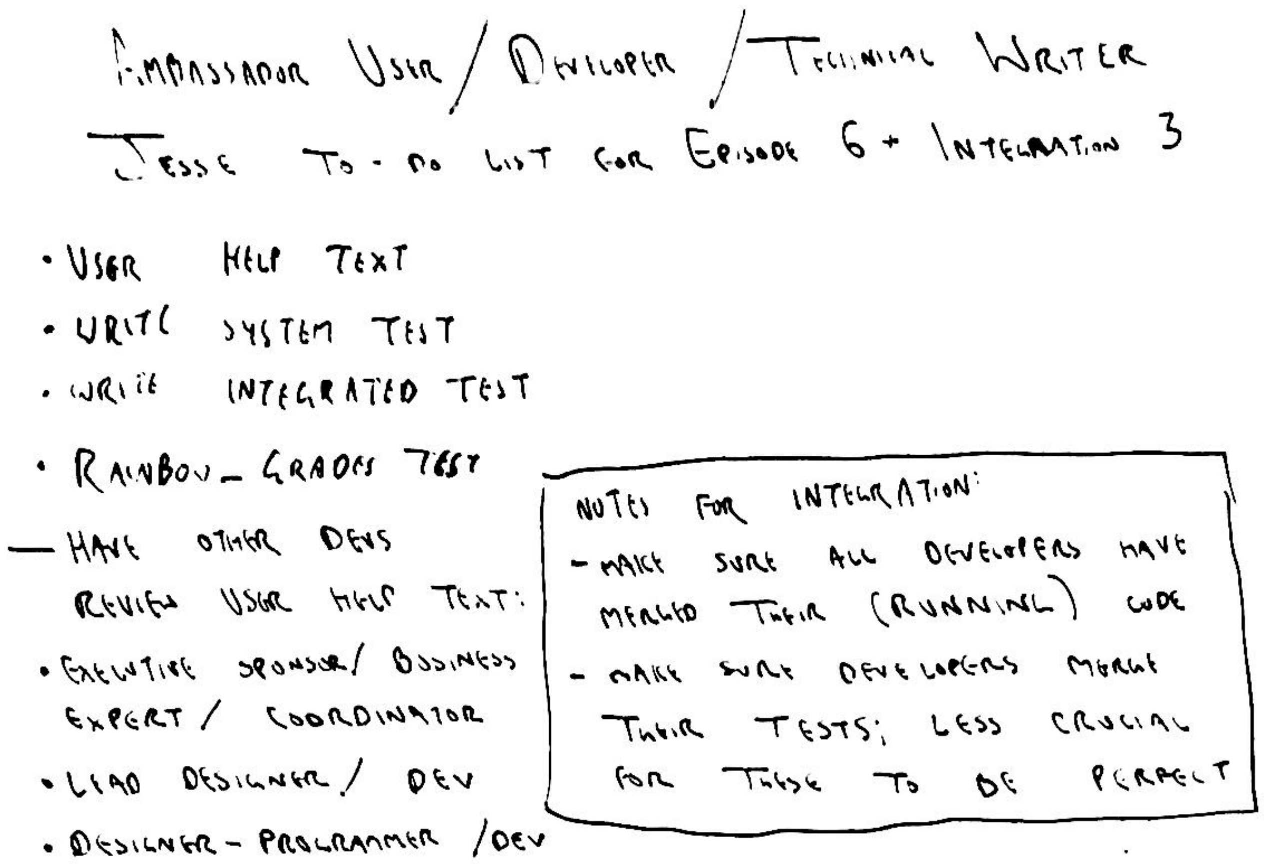
- User Help Text

When to Use Crystal

Crystal is best suited for autonomous teams working to create a final project together. There is also a necessity for developers and other team members to be working in the same space, to enable Osmotic Communication. It lends flexibility and freedom to developers and ultimately ensures products are delivered on time to customers. However, the team must be comprised of skilled individuals and work well as a unit; otherwise, this may be a confusing agile methodology to implement. There is also no large stress on documentation, which may cripple large projects meant for clients to have a thorough knowledge of the product.

Example Artifacts

Example Ambassador User / Developer / Tester / Technical Writer To-Do List:



AMBASSADOR USER / DEVELOPER / TECHNICAL WRITER

JESSE TO-DO LIST FOR EPISODE 6 + INTEGRATION 3

- USER HELP TEXT
- WRITE SYSTEM TEST
- WRITE INTEGRATED TEST
- RAINBOW_GRADES TEST

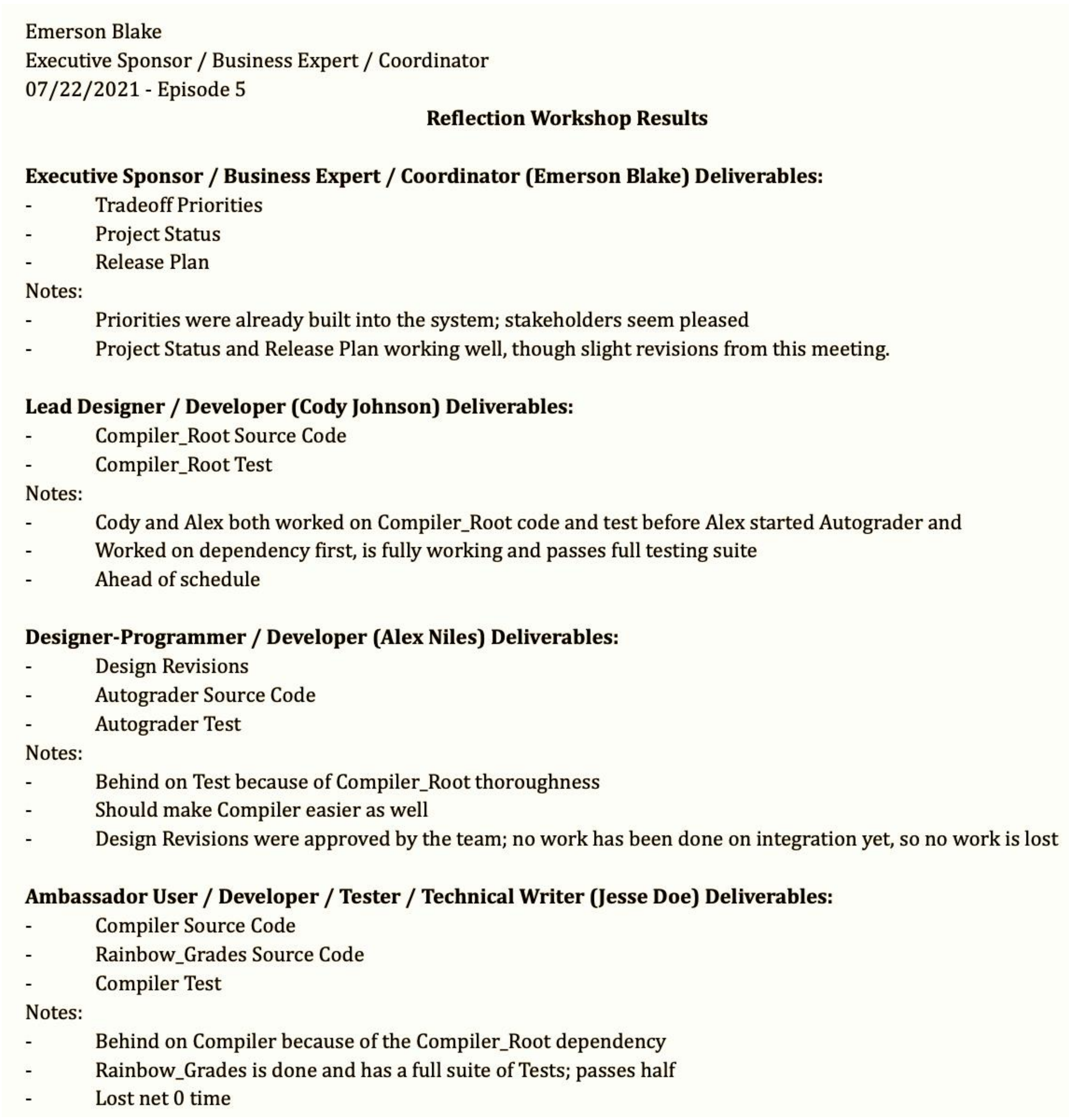
— HAVE OTHER DEVS REVIEW USER HELP TEXT:

- EXECUTIVE SPONSOR/ BUSINESS EXPERT / COORDINATOR
- LEAD DESIGNER / DEV
- DESIGNER - PROGRAMMER /DEV

NOTES FOR INTEGRATION:

- MAKE SURE ALL DEVELOPERS HAVE MERGED THEIR (RUNNING) CODE
- MAKE SURE DEVELOPERS MERGE THEIR TESTS; LESS CRUCIAL FOR THESE TO BE PERFECT

Example Executive Sponsor / Business Expert / Coordinator Reflection Workshop Results:



Emerson Blake
Executive Sponsor / Business Expert / Coordinator
07/22/2021 - Episode 5

Reflection Workshop Results

Executive Sponsor / Business Expert / Coordinator (Emerson Blake) Deliverables:

- Tradeoff Priorities
- Project Status
- Release Plan

Notes:

- Priorities were already built into the system; stakeholders seem pleased
- Project Status and Release Plan working well, though slight revisions from this meeting.

Lead Designer / Developer (Cody Johnson) Deliverables:

- Compiler_Root Source Code
- Compiler_Root Test

Notes:

- Cody and Alex both worked on Compiler_Root code and test before Alex started Autograder and
- Worked on dependency first, is fully working and passes full testing suite
- Ahead of schedule

Designer-Programmer / Developer (Alex Niles) Deliverables:

- Design Revisions
- Autograder Source Code
- Autograder Test

Notes:

- Behind on Test because of Compiler_Root thoroughness
- Should make Compiler easier as well
- Design Revisions were approved by the team; no work has been done on integration yet, so no work is lost

Ambassador User / Developer / Tester / Technical Writer (Jesse Doe) Deliverables:

- Compiler Source Code
- Rainbow_Grades Source Code
- Compiler Test

Notes:

- Behind on Compiler because of the Compiler_Root dependency
- Rainbow_Grades is done and has a full suite of Tests; passes half
- Lost net 0 time

Schedule

In the scenario where there are only four developers, the Crystal Clear methodology is in effect. This means there are a number of overlapping and missing roles. The Executive Sponsor in this project also takes on the roles of the Business Expert and Coordinator because this greatly reduces the need for all three of these roles to communicate with one another. They are the main authority when it comes to "selling" the product, and are the de facto "leader" of this group. The remainder of the roles are also developers: we have the Lead Designer / Developer, Designer-Programmer / Developer, and the Ambassador User / Developer / Tester / Technical Writer. This way, the team is divided by amounts of work and specialty, which is good for communication in the long run.

First Three Episodes:

	Episode 1	**Episode 2 + Integration 1**		**Episode 3 + Reflection Workshop**
Roles	**Days 1 - 3**	**Day 4**	**Day 5**	**Days 6 - 7**
Everyone	Team Structure + Conventions	-	-	Reflection Workshop Results
Execuitive Sponsor / Business Expert / Coordinator	Risk List + Mission Statement	Revised Risk List + Iteration Plan & Status	Project Status	Project Map
Lead Designer / Developer	Architecture Description	-	Design Feedback	Architecture Revisions
Designer-Programmer / Developer	Design Sketches + Notes	Screen Drafts	Architecture Feedback	Design Revisions
Ambassador User / Developer / Tester / Technical Writer	User Role Model	Actor-Goal List	-	Actor-Goal Revisions

Each field within the table is an assignment for the person whose role is within the corresponding row; each assignment should be *started* on the assignment's corresponding column. Notice how there are grey fields filled in; those days are not for rest. Rather, there is nothing new assigned on that day, so each role must work on assignments already received. Notice also that Episode 2 has an Integration; this necessitates a company-wide meeting about how to integrate all current parts, and discussions of what has been created over the last two episodes, and concludes with all parts folding into the main work in progress. All integrations are followed by Reflection Workshops, where all company members gather together and talk about what worked and what didn't work in the last episode. New ideas are tossed around here, as are reviews of work done so far. Note how many revisions should be started upon having the Reflection Workshop.

Next Three Episodes:

	Episode 4 + Integration 2		**Episode 5 + Reflection Workshop**	**Episode 6 + Integration 3**	
Roles	**Days 8 - 9**	**Days 10 - 11**	**Days 12 - 15**	**Days 16 + 17**	**Days 18 + 19**
Everyone	-	-	Reflection Workshop Results	-	-
Execuitive Sponsor / Business Expert / Coordinator	Tradeoff Priorities	Project Status	Release Plan	Viewing Schedule	Project Status
Lead Designer / Developer	Compiler_Root	-	Compiler_Root Test	-	Common Domain Model
Designer-Programmer / Developer	Autograder	-	Autograder Test	-	-
Ambassador User / Developer / Tester / Technical Writer	Compiler	Rainbow_Grades	Compiler Test	User Help Text	Write System Test + Integrated Rainbow_Grades Test

This is the meat of the project; notice how the creation of source code finally begins on days eight, nine, and ten, with Compiler_Root, Autograder, Compiler, and Rainbow_Grades all assigned to different developers and given multiple days just to code. Then testing begins immediately after; there will be overlap between coding and testing, though coding does not need to be finished for testing to begin. It is important to test frequently to maximize efficiency. The Executive Sponsor, meanwhile, checks in with all of the developers at every state of creation, drafts Project Statuses to be created as a result of every Integration, and reads at every Reflection Workshop. This keeps development on track and the leader adequately involved with the details of development.

The Final Stretch:

	Episode 7 + Reflection Workshop	Episode 8	Episode 9 + Integration 4	Final Episode (10) + Final Integration (5) + Final Reflection Workshop
Roles	**Day 20**	**Day 21**	**Days 22 - 23**	**Day 21**
Everyone	Reflection Workshop Results	-	-	Reflection Workshop Results and Final Reflection Results
Execuitive Sponsor / Business Expert / Coordinator	Use Cases + Requirements File	Risk, Map, Plan, Schedule Revisions	Project Status	Final Evaluation
Lead Designer / Developer	Test Update	Integrated Compiler_Root Test	Integrated System Test	Packaged System
Designer-Programmer / Developer	Test Update	Integrated Autograder Test	Integrated System Test	Packaged System
Ambassador User / Developer / Tester / Technical Writer	User Help Text Feedback	Integrated Compiler Test	Integrated System Test	Packaged System

Episodes seven through ten are filled with testing on the developer side, and communication and evaluation on the Executive's side. Here, they will gauge whether or not more time must be added to the timeline for testing, check if the developers are on track for delivering their promised product within the clients' specifications, and add documentation as needed. After the final evaluation on the last day, there should be a shippable product ready for the use of students and professors at RPI. It is assumed that everyone on the team would receive a Submitty sticker and quarter-zip sweatshirt in thanks.

Question C

Submitty is a large open source project and has, therefore, had a number of different development groups of varying sizes. In this way, it differs from some other projects that might do well with the Crystal Agile framework; projects with concrete deadlines and paying customers might be better suited for this framework. However, a Crystal Clear schedule with four full-time workers should be sufficient to complete the two new features I have outlined on time and within spec. The developers have twenty-one days to execute the assigned task. There are five questions to ensure that we stay on track: 1) What are your current risks, and how have you prepared for them? 2) How much time have you allocated for testing, and how do you justify that amount of time? 3) How are tasks divided within the team? 4) How is the "manager" going to encourage honesty, as well as productivity, during reflection workshops? 5) How are late dependency issues minimized?

Our current risks are caused by the Crystal methodology and our small development window; the timing of this project is a risk by itself, so that will be discussed later. The value of robust communication within the Crystal methodology cannot be overstated. Because of the lack of written material and the emphasis on human interaction, the project hinges entirely on talk between teammates. To remediate the risk of garbled communication, we will have meetings before work starts every day, reflection workshops, and develop in the same space if that is possible. If we have remote team members, then we should have a call running during the work day, so osmotic communication can still happen. Luckily, if communication fails, this should be caught early in a reflection workshop, and shouldn't set the team back time-wise by more than an episode. However, if communication fails completely, then roles may have to be redistributed. In Mythical Man Month, a large onboarding time is factored into the development process, which sets back progress exponentially. Moving roles within our team should not take more than a day off of the schedule due to the Crystal methodology; training people and firing others would be a last resort. The Crystal methodology creates and partially remediates the risk of faulty communication by design; therefore, it is our job to be very careful and deliberate about this necessary piece of development.

Another question that might be asked to show whether or not the Submitty team will deliver the new features on time would be how much time we have allocated for testing. The author of Mythical Man Month recommends that, "half of the schedule [be] devoted to debugging of completed code" (Brooks, 5). As a result, testing starts on day twelve out of the twenty-one day development period, which is a bit more than half of the whole timeline. This estimate counts on the source code being fully implemented in four days; not many new classes or dependencies need to be created to implement the features outlined in Question A, so this is more than reasonable. Additionally, the people writing new features will be responsible for testing their own code, which will further streamline the process of testing. The risk of not allocating enough time for testing is huge, which lends value in the question of whether or not the product will be implemented in time. The extended amount of time allocated for testing should be more than sufficient. Though should it run over time, the whole project's deadline would need to be pushed out to accommodate.

Even with the Crystal Clear development method, each team member will have to take on multiple "roles" as are defined in the Crystal methodology. The question of how those tasks should be divided is a valuable one to ask, considering that an unequal distribution of work among group members will invariably lead to inefficiency and lateness. With these two features, tasks were divided among four developers by specialty. For example, there is one "Business Person" who takes on the tasks of the Executive Sponsor, Business Expert, and Coordinator. The Lead Designer and Designer-Programmer have additional development duties, in addition to their assigned deliverables based on Crystal. There is also a "Tester" who is in charge of the Ambassador User, Developer Tester, and Technical Writer deliverables. Jobs were decided based on concentrations and what would go best together; by having one "business person" and three development-heavy roles, no one person has to overextend themselves by learning the tricks of multiple trades. If the business and designer roles were distributed equally among group members, their work may not otherwise be as focused.

Managers need to exhibit restraint in punishing lackluster status reports at reflection workshops. This is how powerful team members within the project encourage honest reflections and productive episodes from developers. The value of this is intrinsic; true documentation within the development process encourages managers to help if they can, and redistribute problems according to who can best solve them. The Submitty team solves this by developing in a shared space and encouraging osmotic communication. As the "Business Person" is updated on the development process in real time, they will not need to wait until a reflection workshop to help out. Instead, because the whole team is together all the time, help can be assigned to each issue as it comes up. This is in direct contrast to more traditional business models, where employees work from home or in an office in separate spaces and meet once a week (or even less frequently) with their superiors. Errors are not caught often enough in the latter scenario, and the whole team ultimately suffers for it.

Solving late dependency issues early is important. Such problems push projects farther from their due date by encouraging developers to avoid personal responsibility for their deadlines. The Submitty team avoids these issues by keeping development as modular as possible. The classes created in each new feature can be created independently without an excess of communication, which decreases dependencies and thus the possibility that one such dependency will be late. Adhering to the Open-Closed Principle keeps the code base clean and enables this modularity. The consequences of not doing this, despite having a messy code base, creates fragility and other code smells within the system, which makes development harder for individual developers. Good coding practices are good for keeping production teams efficient, as well as writing good programs.

References

Mythical Man Month:

https://sites.google.com/site/rpisdd/resources

Caleb Esquino and Sam Stuart’s Presentation:

https://drive.google.com/file/d/1B3JmVIcBa80

https://docs.google.com/presentation/d/19joWXPOjHKyEKQS-OnqNz5AW1JJBLmmzjURZrjrTWEA/edit#slide=id.p

Submitty Development Website:

https://submitty.org/index/overview

Additional Crystal Resources:

https://airfocus.com/glossary/what-is-the-crystal-agile-framework/

https://www.productplan.com/glossary/crystal-agile-framework/

https://www.tuannguyen.tech/2019/07/software-development-methodology-crystal/

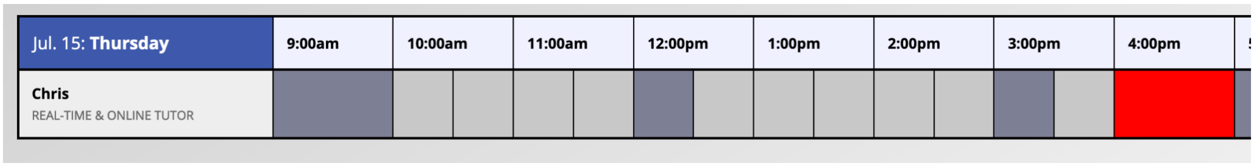
https://www.toolsqa.com/agile/crystal-method/

Submitty Repository:

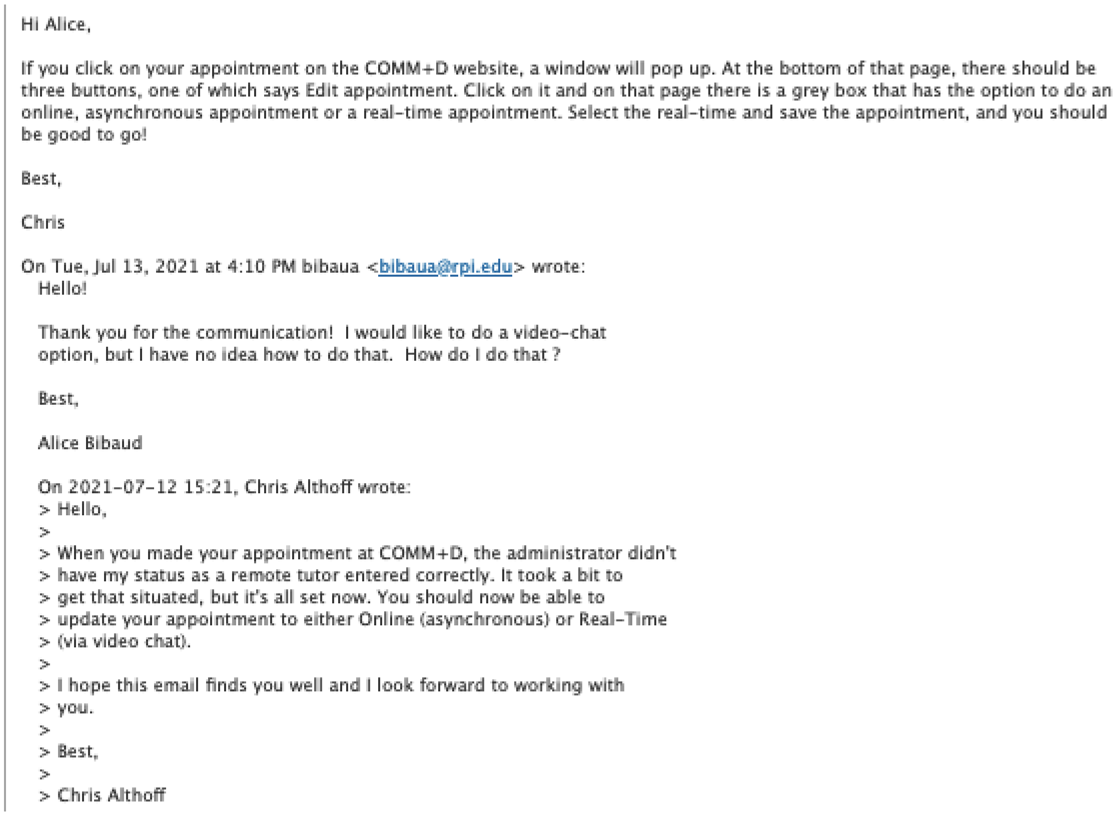
https://github.com/Submitty/Submitty

Proof of COMMD Meetings

First Meeting:



Jul. 15: Thursday	9:00am	10:00am	11:00am	12:00pm	1:00pm	2:00pm	3:00pm	4:00pm
Chris REAL-TIME & ONLINE TUTOR	⬛	⬜⬜	⬜⬜	⬛⬜	⬜⬜	⬜⬜	⬛⬜	🟥



Hi Alice,

If you click on your appointment on the COMM+D website, a window will pop up. At the bottom of that page, there should be three buttons, one of which says Edit appointment. Click on it and on that page there is a grey box that has the option to do an online, asynchronous appointment or a real-time appointment. Select the real-time and save the appointment, and you should be good to go!

Best,

Chris

On Tue, Jul 13, 2021 at 4:10 PM bibaua <bibaua@rpi.edu> wrote:

Hello!

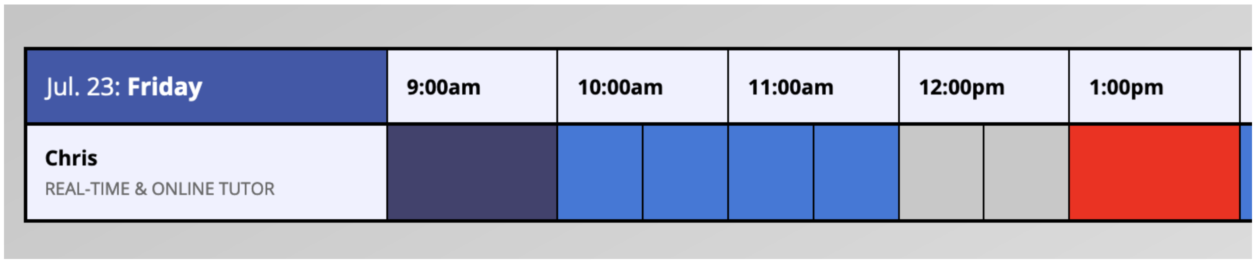
Thank you for the communication! I would like to do a video-chat option, but I have no idea how to do that. How do I do that ?

Best,

Alice Bibaud

On 2021-07-12 15:21, Chris Althoff wrote:
> Hello,
>
> When you made your appointment at COMM+D, the administrator didn't
> have my status as a remote tutor entered correctly. It took a bit to
> get that situated, but it's all set now. You should now be able to
> update your appointment to either Online (asynchronous) or Real-Time
> (via video chat).
>
> I hope this email finds you well and I look forward to working with
> you.
>
> Best,
>
> Chris Althoff

Second Meeting:



Jul. 23: Friday	9:00am	10:00am	11:00am	12:00pm	1:00pm
Chris REAL-TIME & ONLINE TUTOR	⬛	🟦🟦	🟦🟦	⬜⬜	🟥

I asked the professor if this was sufficient, and he said yes. Please let me know if you have any questions!