The Icarus Protocol

Project Testing and Acceptance Plan

Lincoln Middle School

Pluribus Doctrina

Collin Nelson;

Anna Ueti;

10.28.22

TABLE OF CONTENTS

I. Introduction

I.1. Project Overview

Icarus Protocol is a game being made in collaboration with Lincoln Middle School, designed to teach basic Python programming skills to students with little to no prior programming experience. The game is designed to be fun and engaging, focusing on the key design objective of “putting fun first” to create an experience that increases knowledge retention and student enjoyment by allowing them to develop their skills through an activity that feels more like a game than like a series of lessons. In the game students will play the role of a ship AI, trying to repair the broken software systems of the ship by correcting the ship’s incorrect code. Doing this introduces and teaches a variety of programming concepts and challenges the students to extend their knowledge through optional challenges and boss levels. The students will repair the code by writing real snippets of Python code, which will be executed dynamically at runtime and evaluated in real time to determine if the player’s code successfully solves the problem at hand.

The game is being constructed in the Unity Engine, which imposes some particular challenges for testing and validating the game scripts. We consider automated tests crucial for the core IronPython integration that handles the ability of the game to run user code dynamically at runtime, as well as the save/load code that manages player save data. Most other scripts would be good to test, but mostly stand to interact with UI elements or manipulate objects within the Unity game engine, making them both less necessary, and more difficult, to test.

I.2. Test Objectives and Schedule

The objective of tests in this kind of project is generally to gain confidence that the code we’ve written will successfully and consistently execute the desired behavior and fulfill all requirements at runtime. Automated tests especially serve the additional purpose of serving to safeguard the project against being broken by future updates. Our automated and manual tests will accomplish both of these purposes, helping to ensure that all core features are operating correctly each time that we deploy a new build of the game. We feel this is especially important to have established as we move into the second half of our development process, which will involve directly interacting with users in a series of beta-tests to receive feedback and rapidly iterate on our build. High quality tests now ensure that our iteration later can be more agile and more reliable.

The project being constructed in Unity leads to us needing to adopt some very specific testing technologies. For unit testing we will have to use the Unity Test Framework (Unity Technologies), a special C# unit testing framework designed to work with and within the Unity Engine. We can also use this framework to develop some isolated level of integration testing, although full integration testing will be utterly impossible because most of our code interacts with the external Unity Engine systems that we have no reason to do integration tests on. Where applicable, some integration tests will be used to ensure that closely related components created entirely by us are in fact integrating properly. We will also need to create and document extensive manual testing procedures. Most of our system testing will have to be done manually, due to the lack of automated tools for testing compiled Unity executables. This will have to include manual tests run by ourselves and during the acceptance testing process to ensure that the product performs all expected functionalities as intended when interacted with by a user, as well as procedure for performing formalized playtests with middle schoolers, gauging their engagement, the ease of use, and receiving feedback from them on their experience.

Our testing process will deliver 3 major deliverables. The first is a suite of unit tests covering all non-trivial non-UI scripts produced when creating the functionality of the application. The second is a document containing the testing procedure including operational instructions and expected results for all manual functional tests and playtests, and the final deliverable is a github CI pipeline created using the GameCI framework for running our pull-requests through a tested CI pipeline (GameCI).

It should be noted that the GameCI framework is a third party tool not associated with the Unity Engine. We also have yet to confirm the viability of this as an approach. Unity's built-in CI system is unfortunately a paid feature that we aren't prepared to pay monthly for. GameCI seems promising and professionally developed, but we may encounter problems that make the use of this system untenable. If these types of issues (package conflicts, licensing issues, etc.) occur we may have to opt to not use CI in our project as a result. This would obviously not be ideal, and it isn't our intention to go this route, but Unity-enabled CI is still a problem that is actively being solved, and all existing solutions have their own flaws and concerns.

I.3. Scope

This document discusses our plans for how we intend to test and create test material for Icarus Protocol. It will cover our general testing and deployment process, as well as our testing plans for various forms of essential product testing. This document will not, however, cover the specific tests we will write, the content of those tests, or the exact content of any functional test documents or playtest feedback forms, etc.

II. Testing Strategy

Project testing will be designed to create full automated tests of all core functionality of the game, running these tests through a CI pipeline for continuous integration. The major components of the core functionality are the IronPython integration, and the Save/Load system. Smaller nonessential or non-core functionality may be tested, but due to time constraints may be allowed to run without automated tests. Because Unity automatically handles many errors during runtime without interrupting the flow of the game, errors in these components will not compromise the overall operation of the finished system. Our specific testing procedure is broken down below into 2 distinct processes. The first process describes how we develop, run and push code tested through automated unit/integration testing, as well as manual FT and acceptance testing. The second process is a preliminary outline of our process for performing unit tests, to be finalized later with the help of Lincoln Middle School as we move into the alpha and beta testing phases of the project.

Developer Testing Process

1. **Developer Writes Code:** We do not intend to use TDD for this project. As such, we begin by creating and attaching the feature implementation to the Unity Project.

Implementation is considered complete one the feature appears to successfully accomplish all required functionality.

2. **Determine Test Cases:** For each core functionality, we will create at least 2 test cases. One tests the ordinary expected operation of the unit, and the other tests exceptional or invalid behavior. Nonessential or small functionality, if tested, is acceptable if it only includes a test case for its ordinary behavior.
3. **Run Tests:** The developer should run the tests on the code, running all tests currently in the project, including the ones that they have recently added.
4. **Fix Issues:** If any of the tests fail in **Step 3**, the developer should identify the source of the failure and repair the bugs, rerunning tests as needed until all tests pass.
5. **Developer Pushes Code To Remote:** The developer will push their code to the remote. Assuming that we are successfully able to establish a Unity-compatible CI pipeline, the push will be run through this pipeline and results will be shown. A branch is only eligible for merge if its most recent commits pass all tests run in the CI
6. **Developer Makes a Pull Request Against Main:** The developer makes a PR against the main branch, and adds at least one other developer as a review. The branch should not be eligible for merge unless and until the reviewer(s) perform a code review and provide approval.
7. **Merge Branch:** The branch will be merged into main assuming it passes all previous steps, and the CD pipeline will deploy a new release of the game to github, if applicable.

Playtesting Process

1. **Create Playtest Build:** A developer will create a build of the game set up for the playtest. This could be a fresh build of the game, or a modified build designed to omit or skip certain content to allow testers to skip to relevant playtest material.
2. **Deliver Build & Allow Time For Testing:** With the build delivered, a time period potentially of several weeks should be allowed for students to test the material.
3. **Deploy Feedback Forms:** As testers complete the content under test, they should be prompted to fill out a form. This form should ask questions designed to directly and indirectly gauge their interest in the game, the amount of fun they had playing it, the amount they felt that they learned, and whether they are retaining knowledge from the game.
4. **Collate Feedback Data:** Feedback should be collated into a feedback record document, and results should be analyzed for useful insights into what can be changed or improved for the next iterations.
5. **Make Any Necessary Changes:** Based on the analyzed feedback, make changes to the content, UI, or player experience to improve the experience for the next iteration of playtesting.

III. Test Plans

III.1. Unit Testing

The team will generally follow traditional unit testing procedures, however where the team's unit testing methodology will diverge in areas that are Unity Engine specific. The team will be using the Unity Test Framework, formerly known as the Unity Test Runner, as the avenue of testing for this project (Unity Technologies). In order to consider the code sufficiently tested, the team will be required to test all core game functionalities. A core game functionality is defined as a game object functionality that would render the game unplayable if non operational, it is non-trivial and a non-UI script. Additionally, the team will evaluate the relevance and impact of all non-essential units. If the developer feels it necessary, more unit tests may be developed. These units under review would be scripts pertaining to game objects that are highly frequented or provide essential user functionality, but are not considered core game features. For this section, we will defer to individual developer discretion when considering the extent of additional unit tests.

III.2. Integration Testing

Given the nature of working within the Unity Test Framework, our integration testing will mirror the Unit testing section however with more limitations. Due to requirements of the Unity Engine, our team must diverge from standard integration testing processes, testing isolated code clusters without the ability to fully integrate the application. Additionally, the unique structure of the project may lead to developer discretion in whether or not to attempt to integrate less amenable script structures.

III.3. System Testing

III.3.1. Functional testing:

The team's functional testing plan is primarily reliant on manual testing implemented and tested by the developers. For this section, the team will predominantly focus on the previously outlined Requirements and Specifications document, which contains developer and stakeholder expectations for project functionality. Each functional requirement, outlined in the document mentioned above, will be associated with one functional test. This section of the document is subject to revision if there is an update to or restructuring of the project's functional requirements. Given the nature of working with the Unity Engine, these tests will be manually validated by the developers. In the case that a functional test should fail, the developer who is testing that component will provide a description of the test conditions and request the original developer to reevaluate and solve the error.

III.3.2. Performance testing:

Leaning into the strengths and unique qualities of the Unity Engine, our team is opting to use the Unity Profiler tool (Unity Technologies). The tool measures and provides performance information about the application in areas such as the CPU and memory. This is especially important for the Icarus Protocol game given poor performance doesn't reflect well on the gameplay functionality of the application. Additionally, this tool will provide the resources for the developers to monitor the performance of the application under variable loads. In regards to the non-functional requirements specified in the Requirements and Specifications document, our team will manually test the performance of these aspects. Given the non-functional tests rely on qualitative metrics rather than quantitative, developer discretion will be given.

III.3.3. User Acceptance Testing:

In collaboration with Lincoln Middle School, the team will employ play testing for several iterations before the final release of the game is completed. Our testing strategy for user acceptance testing, will be outlined below with sections for provided resources, instructions for testing groups, and plans for feedback and revision. This outline mirrors the Playtesting Process detailed above, however this description focuses more on the developer view.

A. Resources
    a. A build of the application (for each testing student)
    b. A google form for feedback (for each testing student)
    c. A google form for teacher feedback (for testing facilitator)
B. Instructions
    a. A small select group of students will be chosen for this testing iteration.
        i. Note: a student will only be able to participate in a testing group once.
    b. Each student in the testing group will be provided a working build of the Icarus Protocol game.
    c. A to be determined timeframe will be provided for testing.
    d. After the student finished the game or by the end of the testing time frame, the student will be provided an exit google form for feedback.
    e. After all students have completed the game or the duration of the testing time frame has expired, the facilitator will be provided an exit google form for feedback.
C. Revision
    a. The team will receive and review all feedback forms
    b. The team will then conviene and deliberate on modifications to pursue based on the testing group feedback.
        i. If a new feature is pursued or a bug is found, the team will provide documentation for the addition or modification and will continue with updating the project accordingly.

*This outline is intended to be reiterated for testing purposes.

The final iteration of the user acceptance testing will involve positive responses from the students and facilitator, as well as positive skill growth from the beginning of the game to the end and demonstration that all required functionality is functional in the final build.

IV. Environment Requirements

Our testing environment is predominantly self-contained with Unity Engine proprietary tools. As mentioned above, both unit and integration testing will use the Unity Test Framework and their testing strategies will generally mirror each other.

However, the team will be using an external component, GameCI if possible, in conjunction with the Github CI pipeline. This framework will provide a testing pipeline that will allow developers to guarantee that tests are run and passed before committing and merging work.

There are no specific hardware requirements.

V. Glossary

alpha-test: The preliminary testing stage that validates that the Icarus Protocol performs as expected.
beta-test: The final testing stages that ensure the Icarus Protocol is qualified for final release.
CI/CD: Continuous Integration/ Continuous Deployment. This provides the team a method to ensure tests are passing as software is updated by developers.
functional requirements: The requirements for the Icarus Protocol that can be tied to a software functionality implemented by the team.
non-functional requirements: The requirements for the Icarus Protocol that are general, often qualitative, and not related to a specific functionality.
non-trivial: This refers to methods that have core game functionality that the game requires for complete
TDD: Test Driven Development. A type of software development characterized by writing tests first.

VI. References

n/a, "GameCI," *GameCI · The fastest and easiest way to automatically test and build your game projects*. [Online]. Available: https://game.ci/. [Accessed: 28-Oct-2022].

U. Technologies, "Profiler Overview," *Profiler overview*. [Online]. Available: https://docs.unity3d.com/Manual/Profiler.html. [Accessed: 28-Oct-2022].

U. Technologies, "Unit testing," *Unity*. [Online]. Available: https://docs.unity3d.com/Manual/testing-editortestsrunner.html. [Accessed: 28-Oct-2022].