2.1 Introduction

The introduction puts the test activities described in the project document in the context of the overall project and test effort for the project.

2.1.1 Project information

Mindustry is a factory-building game initially released on September 26th, 2019 with tower defense and RTS elements. It allows one to create elaborate supply chains to feed ammo into turrets, produce materials to use for building, and construct units. Command units to capture enemy bases, and expand your production. Defend your core from waves of enemies. Mindustry was developed and published by Anuken, runs on the libGDX engine, and is available on Android, iOS, Microsoft Windows, Linux, and Mac operating systems.

- mary

2.1.3 Scope

The scope should describe what is being tested for this document level. Details about the portion of the software being tested should be included.

1. Creation of Control Flow Graphs (CFGs)

   This pertains to identifying the basic blocks for the methods which have at least one branching decision in them.

2. Achieve Edge Coverage

   This will involve selecting the test paths from the Control Flow Graphs (CFGs) drawn above so that the paths achieve edge coverage.

3. Generating Test Cases

   Upon selecting all test paths to achieve edge coverage for the Control Flow Graphs (CFGs), test cases will be generated. Each test case will include the test data i.e. the input data, and the expected output for that test path. Each test case will begin with the start node, i.e. the 0th node and ends with the end node, as per its corresponding CFG. The main method should achieve program wide edge coverage and abstract complex test paths into simple test paths.

4. JUnit testing

   This step will involve utilizing JUnit to implement and execute the generated test cases and find out the faults each. Each test method will be checked to confirm edge coverage is present.

5. Branch Coverage Report

We will be using the IDE of Eclipse to generate Branch Coverage Reports based on the JUnit test. This will aid in visualizing which branches from the program will be covered by the tests and which will not be. This information will also be used to potentially increase the coverage percentage by covering the uncovered branches/lines of code.

2.1.4 Reference

Related documents should be referenced here. External and internal documents should be identified and listed separately.

Github: https://mindustrygame.github.io/
Trello: https://trello.com/b/aE2tcUwF/mindustry-trello
Anuke: https://anuke.itch.io/mindustry
Discord: https://discord.com/invite/mindustry
Wiki: https://mindustrygame.github.io/wiki/
App store: https://apps.apple.com/us/app/mindustry/id1385258906?ign-mpt=uo%3D8
https://f-droid.org/packages/io.anuke.mindustry/
https://play.google.com/store/apps/details?id=io.anuke.mindustry&pli=1
https://mindustry.fandom.com/wiki/Mindustry_Wiki

2.2 Details for The Level Test Plan

This section discusses the environment requirements and highlights the approaches used to test the functionality of this application.

The system under test is the Mindustry Software.

- Requires Java, JDK 16-17
- Open a terminal in the Mindustry directory and run specific commands based on the operating system in use:
    - Windows:
        - Running: gradlew desktop:run
        - Building: gradlew desktop:dist
        - Sprite Packing: gradlew tools:pack
    - Minux/Mac OS
        - Running: ./gradlew desktop:run
        - Building: ./gradlew desktop:dist
        - Sprite Packing: ./gradlew tools:pack
- Common Issues & How to Troubleshoot:
- Permission Denied
- If the terminal returns Permission denied or Command not found on Mac/Linux, run chmod +x ./gradlew before running ./gradlew. This is a one-time procedure.
- Gradle may take up to several minutes to download files. Be patient.

- After building, the output .JAR file should be in /desktop/build/libs/Mindustry.jar for desktop builds, and in /server/build/libs/server-release.jar for server builds.

2.2.1 Test items and their identifiers

Test Item: Game Install via App Store and Steam
Test Identifier: T1
Description: The test cases verifies the functionality of installing the game. The game should be installable from the app store according to the minimum app store version required listed in the app store description. Similarly with Steam. Moreover, the game should be open and load resources successfully upon install.

Test Item: Login & Signup
Test Identifier: T2
Description: Functionality test. User should be able to sign up with username and password or through Google signin. Upon signin, the user should be able to view the rules and map of the game.

Test Item: View Map
Test Identifier: T3
Description: Functionality test. User should be able to start the game on a new map after signing in.

Test Item: View Rules
Test Identifier: T4
Description: Functionality test. User should be able to view the rules of the game after singing in.

Test Item: Enemy Attack
Test Identifier: T5
Description: Functionality test. The user should be able to view enemies and receive damage to player's structure upon enemy attack.

Test Item: Build
Test Identifier: T6
Description: Functionality test. The user should be able to build towers and conveyor belts. The user should be able to use any of the materials and methods provided in the game

Test Item: Defend
Test Identifier: T7
Description: Functionality test. The player should be able to fire at the enemy using ammunition, and the enemy should successfully go down after the HP bar reaches 0.

Test Item: Special Items
Test Identifier: T8
Description: Functionality test. The player should be able to successfully use items such as powered shields and regeneration projectors to establish their defenses

Test Item: Liquids
Test Identifier: T9
Description: Functionality test. The player should be able to use in-game liquids, such as coolant and lubricant, to fight fire outbreaks and enemy raids

Test Item: Automatic Base Management
Test Identifier: T10
Description: Functionality test. The player should be able to automate base and enemy management successfully.

Test Item: User Quits Midway
Test Identifier: T11
Description: Functionality test. The player's progress should be saved if the player quits midway.

Test Item: Performance
Test Identifier: T12
Description: Non Functional Test. The player should be able to see the result of an action immediately. The user should be able to view enemies and receive damage to the player's structure upon enemy attack within a short time frame.

Test Item: Reliability
Test Identifier: T13
Description: Non Functional Test. The system should not shutdown without being explicitly indicated by the user.

2.2.2 Test traceability information

Test ID	**Test Item**	**Test Origin**
T1	Game Install via App Store and Stream	Functionality Req 1.1 (successful install)
T2	Login & Signup	Functionality Req 2.1 (login) and 2.2 (signup)
T3	View Map	Functionality Req 3 and Design Element (Game cannot be played w/o the map)
T4	View Rules	Functionality Req 4 and Design Element (New users need to see rules to play. Must have proper format)
T5	Enemy Attack	Functionality Req 5 and Design Element (Game cannot be played without enemies, entities

		must move properly)
T6	Build	Functionality Req 6 and Design Element (Game cannot be played without structures, must be formatted in-game properly)
T7	Defend	Functionality Req 7 and Design Element (Defense is critical to the game)
T8	Liquids	Functionality Req 8 and Design Element (Liquids are part of game special features and needed for enemy defense)
T9	Automatic Base Management	Functionality Req 9 and Design Element (Automatic base management is an important feature of the game)
T10	User Quits Midway	Functionality Req 10 and Design Element (User progress must be saved or users will stop playing the game)
T11	User Experience with Viewing Enemies	Non-Functional Req 12 and Design Element (User does not experience lag or a particularly slow response. When User wants to View Enemies they see it almost immediately)
T12	Game Start on Computer	Non-Functional Req 13 and Design Element (Game does not close unexpectedly, user must log out or terminate game session to shut down game)

2.2.3 Features to be tested

The following are a list of features taken from the user manual and documentation:

Features and Associated Classes	Description
Factory Building - mindustry.world.blocks	The user should be able to build towers and conveyor belts. The user should be able to use any of the materials and methods provided in the game.
Initialization - mindustry.maps	Users should be able to start the game on a new map after signing in.
Tower Defence - mindustry.world.defense, mindustry.entities	The player should be able to fire at the enemy using ammunition, and the enemy should successfully go down after the HP bar reaches 0.
Tower Defense - mindustry.type.weapons, mindustry.world.blocks	The player should be able to successfully use items such as powered shields and regeneration projectors to establish their defenses.
Tower Defense - mindustry.world.blocks.liquid/heat	The player should be able to use in-game liquids, such as coolant and lubricant, to fight fire outbreaks and enemy raids.
Factory Building and Tower Defence - mindustry.world.blocks	The player should be able to automate base and enemy management successfully.
Initialization - arc.assets	Users should be able to view the rules of the game after signing in.
Tower Defense - mindustry.type.weapons, mindustry.world.blocks, mindustry.entities	The user should be able to view enemies and receive damage to the player's structure upon enemy attack.
Factory Building - mindustry.world.blocks, mindustry.maps	The user should be able to place and build structures on all open areas of the map.

2.2.4 Features not to be tested

Features not to be tested include details about weather and graphics classes. Testing every type of graphic and special effect would take too long and is beyond the scope of the remaining half of this semester. For this reason, the features to be tested have been limited to major game features and those critical to smooth gameplay. Details in those features would be tested appropriately.

2.2.5 Testing approach

Individual functions are to be tested via JUnit unit testing. System testing is used to ensure that the system meets functionality requirements given a set of inputs. The performance is to be tested using the testing tools listed in section 2.3.2. Moreover, security tests can be performed using appropriate security tools. The usability and functionality of the application is to be tested using a combination of JUnit and other Java testing tools listed in this document. Automated testing applies to individual methods and functions where tests are carried out every time a change is made. In terms of test criteria, there should be at least 95% test coverage.

2.2.6 Item pass/fail criteria

Test Item ID	**Pass Criteria**
T1	Game is successfully installed using either method (App store or Steam). 100% tests pass.
T2	Users can register then login successfully. 100% tests pass.
T3	User is able to view the map on the screen. 100% tests pass.
T4	User is able to view the rules, formatted correctly. 100% tests pass.
T5	User is able to take damage from the enemy with no bugs or failures or errors. 95% tests coverage.
T6	Users are able to build structures such as towers without failures or errors. 95% tests coverage.
T7	Users are able to defend structures using methods provided without failures or errors. 95% tests coverage.
T8	Users should be able to use liquids to attack enemies without any faults or failures. 95% tests coverage.
T9	Users should be able to automate base management with no failures or errors. 95% tests coverage.
T10	User progress should be saved regardless of the game environment upon user quit. 100% tests pass.
T11	Game response to user actions, specifically enemy attack, should be quick, <1s lag time. 95% of the

	tests pass.
T12	Game does not close unexpectedly with any of the test cases (no failures). 100% tests pass.

2.2.7 Suspension criteria and resumption requirements

Criteria for Suspension of Testing

- A critical failure is detected which prevents the primary functionality of the application (such as user gameplay, or user login). In this case, the testing should be suspended in favor of immediately debugging the program to resolve the issue.
- If the application undergoes significant changes such that the test cases are no longer an accurate representation of the application. In this case, testing should be suspended and assessed, with the necessary modifications made to the test cases.

2.2.8 Test deliverables

- Test plan outlining the requirements, features, and approach to testing. It also displays the objective.
- Traceability matrix between test cases and requirements
- A document of all executable test cases
- A documents of test results, test coverage, and findings

2.3 Test Management

This section describes what will be done when and who will do them.

2.3.1 Planned activities and tasks

Group Member	Task	Deadline
Fatima Rahman	Identify characteristics and partitions for testing. Help identify functional and non-functional requirements and develop test cases around those.	Test Plan: Feb 27th Testing Midpoint: March 15th Testing Completion: Final week
Afsah Rabbani	Help define test cases and identify boundaries and characteristics. Plan out test management, and help with carrying out tests.	Test Plan: Feb 27th Testing Midpoint: March 15th Testing Completion: Final week
Hamza Iqbal	Identify modules and components in need of a test and help develop a testing strategy, and help with carrying out tests.	Test Plan: Feb 27th Testing Midpoint: March 15th Testing Completion: Final week

Abdulrehman Khan	Help identify testing classes and resources, and define general information needed for testing, and help with carrying out tests.	Test Plan: Feb 27th Testing Midpoint: March 15th Testing Completion: Final week
Everyone	Help develop test cases using chosen characteristics, partitions, boundaries covering all functional and interface-based requirements. Carry out the tests.	Test Plan: Feb 27th Testing Midpoint: March 15th Testing Completion: Final week

2.3.2 Environment and infrastructure

- Hardware: Windows 10 & 11, Mac OS
- Software and Tools: JUnit (4 or 5), Eclipse, Selenium, Mockito, PowerMock, TestNG, JMeter, JMock
- Browser: Chrome
- Result Capturing Tools: Selenium, Jira
- Database: Google Cloud, PostgreSQL
- Privacy and Security Issues: A secure browser and database connection is needed

2.3.3 Responsibilities and authority

Group Member	Task
Afsah Rabbani	Managing, executing, resolving problems
Fatima Rahman	Designing, executing, resolving problems
Hamza Iqbal	Preparing, executing, resolving problems
Abdulrehman Khan	Checking results, executing, resolving problems

2.3.4 Interfaces among the parties involved

Test case execution and evaluation is to be divided equally between the group members (Afsah Rabbani & Fatima Rehman, Hamza Iqbal & Abdulrehman Khan). In the case that an error occurs, whoever is “on-call” at the time should be contacted. Documentation is done by each team member as the tests are being written and executed.

2.3.5 Resources and their allocation

- 8GB RAM or more
- > intel Core i5
- 64-bit operating system
- Should be able to run project with minimal lag

2.3.6 Training

- Basic knowledge of JUnit testing – obtained through labs in this course and online tutorials
- Intermediate knowledge of Java programming language
- Understanding of ISP (Input Space Partitioning) and CFGs (Control Flow Graphs)
- The above and all other required knowledge can be gained through the lecture slides and searching through Google

2.3.7 Schedules, estimates, and costs

Task	Deadline
Preparation, which includes setting up the testing environment, downloading the necessary resources, and establishing the test plan.	Feb 27th
Design of tests, which includes identifying characteristics and boundaries as well as creating CFGs. Designing interface-based and functional tests.	Feb 27th - Test Plan March 9th - Begin testing and refine as we go
Execution of testing, which involves carrying out tests and identifying faults, errors, and failures	March 9th - TBD

2.3.8 Risks and contingencies

- Testing Requirements are found to be lacking or insufficient: using Agile methodology, the test cases and requirements can be modified on a need-to basis
- Insufficient Resources for Testing - Extra testing tools in Java have been added to the test plan in case the need arises
- Time management - Create a schedule to stay on track, conduct weekly meetings, and have a midpoint date

2.4 General Testing Information

Software testing is an essential phase in the development process that makes sure that software complies with all specifications and performs as planned.

It's critical to have a firm grasp of the processes and metrics used in quality assurance in order to test software efficiently (QA). These steps entail making test plans, carrying out test cases, and recording flaws.

Unit testing will be carried out using Junit. The efficiency of the testing procedure can be evaluated using metrics like defect density, test coverage, and defect escape rate. Code coverage testing could be done with Clover. We require a simple method or unique techniques that can intelligently choose test cases from the test case pool so that all test scenarios are covered.

Hence for input space partitioning test inputs, we use Equivalence Partitioning and Boundary Value Analysis testing techniques to achieve this. A glossary is also useful in ensuring clear communication and understanding of terms and concepts related to software testing.

It is important for testing teams to have access to this general information in order to perform their tasks efficiently and effectively, and to ensure that the software meets the highest quality standards before release.

2.4.1 Quality assurance procedures

Quality assurance (QA) procedures are a set of processes and practices that are implemented to ensure that a product or service meets the specified quality requirements. Testing and verification processes are often included in QA procedures in the context of software development and are carried out at various stages of the software development lifecycle. These procedures aim to identify flaws and problems early on in the development process, which can lower the cost and work needed to resolve them later.

Some common QA procedures used in software development include:
- creating test plans and test cases
- executing tests
- documenting defects.

2.4.2 Metrics
This section should describe how testing will be measured and reported.

Test coverage: This metric measures the percentage of code that has been tested. It is usually calculated using tools that track code coverage during testing

Defect density**:** This metric measures the number of defects per unit of code, such as per line of code or per function. This can help identify areas of the code that may require additional testing or development effort

Defect escape rate: This metric measures the percentage of defects that are found after the software has been released to production. A high defect escape rate may indicate that the testing process needs to be improved

Test cycle time: This metric measures the time it takes to complete a testing cycle, from planning to execution to reporting. It can help identify bottlenecks and areas where the testing process can be streamlined

Test pass rate: This metric measures the percentage of tests that pass versus those that fail. It can help identify areas of the code that require additional testing or development effort

Test case effectiveness: This metric measures the effectiveness of individual test cases in finding defects. It can help identify areas where the testing process can be improved or where additional tests are needed.

2.4.3 Tools

Automation Tools:

- Junit
- Clover
- Selenium
- TestNG

Bug tracking tools:

- Jira
- Bugzilla
- Asana
- VSCode Debugger

2.4.4 Test coverage

This section should describe how coverage is measured and how much coverage is required.

Test coverage is a metric used in software testing to measure the extent to which the source code of a software application is tested by a particular test suite. It is calculated by determining which lines of code were executed during the test and comparing that with the total number of lines of code in the application. Here are some commonly used methods for measuring test coverage:

Statement coverage: This method measures the percentage of executable statements that have been executed during the test. It is the simplest and most common method for measuring coverage

Branch coverage: This method measures the percentage of branches (if/else statements) that have been executed during the test

Path coverage: This method measures the percentage of all possible execution paths that have been tested

The particular needs and requirements of the project will determine how much coverage is needed. A larger test coverage rate is generally preferred as it boosts confidence in the application's quality and lowers the possibility of errors getting undetected. Achieving 100% test coverage, though, might not always be feasible or necessary because certain code might be challenging or impossible to test. The optimum level of coverage will vary depending on the application's complexity, the risk of flaws, and the testing resources that are available. Setting a target level of coverage is typically advised, followed by ongoing monitoring and improvement as the project moves forward.

2.4.5 Glossary

**To be added

2.4.6 Document change procedures and history

**To be added

2.5 Approvals

Specify the names, section number, student ID, and titles of all the students (team members) who participated to provide this plan. Place the signature and date beneath it for each team member.

Name	Student ID	Section
Fatima Rahmamn	500892014	3
Afsah Rabbani	500945712	3
Hamza Iqbal	500973673	3
Abdulrehman Khan	500968727	3

Place Signatures: FR, AR, HI, AK
Date: 02/27/2023