

Course Number	COE 891
Course Title	Software Testing and QA
Semester/Year	W2023

Instructor	Dr. Reza Samavi
------------	-----------------

<b>Project No.</b>	<b>1</b>
--------------------	----------

Lab Title	Final Project
-----------	---------------

Submission Date	April 03, 2023
Due Date	April 03, 2023

Student Name	Student ID	Signature*
Fatima Rahman	500892014	F.R
Abdulrehman Khan	500968727	A.R
Hamza Iqbal	500973673	H.I
Afsah Rabbani	500945712	A.R

*\*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work.*

## Objective

The objective of this project is to apply the knowledge of software testing practices on open source projects using the information provided by course COE 891. For this project, the team chose to use Mindustry. 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.

## Findings and results of application testing:

The features tested in the final phase of this project followed that which was outlined in the test plan proposal. The following are a list of features taken from the user manual and documentation:

Table 1: Features to be tested

Feature #	Features and Associated Classes	Description
1	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.
2	Initialization - mindustry.maps	Users should be able to start the game on a new map after signing in.
3	Health and Damage - mindustry.world.defense, mindustry.entities.comp.Health	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.
4	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.
5	Tower Defense - mindustry.world.blocks.liquid/heat	The player should be able to display in-game liquids, such as coolant and lubricant, to fight fire outbreaks and enemy raids.
6	Factory Building and Tower Defence - mindustry.world.blocks	The player should be able to automate base and enemy management successfully.
7	Initialization - arc.assets	Users should be able to view the rules of the game after signing in.
8	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.

9	Factory Building - mindustry.world.blocks, mindustry.maps	The user should be able to place and build structures on all open areas of the map.
---	---	--

## Input Space Partitioning (ISP):

You will need to use Boundary Value Analysis (BVA) to test your input space based on the application of a Java class. Obviously, you will implement/code your testing programs.

Boundary Value Analysis (BVA) is a software testing technique used to identify errors and faults in the boundary conditions of software applications. It is used to test input values that are on the edge of the acceptable range, to ensure that the application behaves correctly.

### Boundary Value Analysis for Factory Building - Mindustry.world.blocks (constructBlock.java)

Boundary Value Analysis (onDestroyed()):

Test Case	Input Value	Expected Result	Actual Result	Pass/Fail
1 (Valid)	Valid Tile object with a block type of "copper-wall" at position (10, 10)	Block state is updated to air and tile is removed, and resources (1 copper, 2 lead) are dropped	As expected	Pass
2 (Lower boundary value)	Null Tile object	NullPointerException is thrown	NullPointerException was thrown	Pass
3 (Lower boundary value)	Tile object with a block type of "air" at position (0,0)	Block state is updated to air and tile is removed, and no resources are dropped	As expected	Pass
4 (Upper boundary value)	Tile object with a block type of "thorium-reactor" at position (100, 100)	Block state is updated to air and tile is removed, and resources (3 thorium) are dropped	As expected	Pass

All test cases pass, which means that the onDestroyed() method handles all possible input values correctly, including edge cases and boundary values, based on the actual values used for testing.

Boundary Value Analysis (updateTile()):

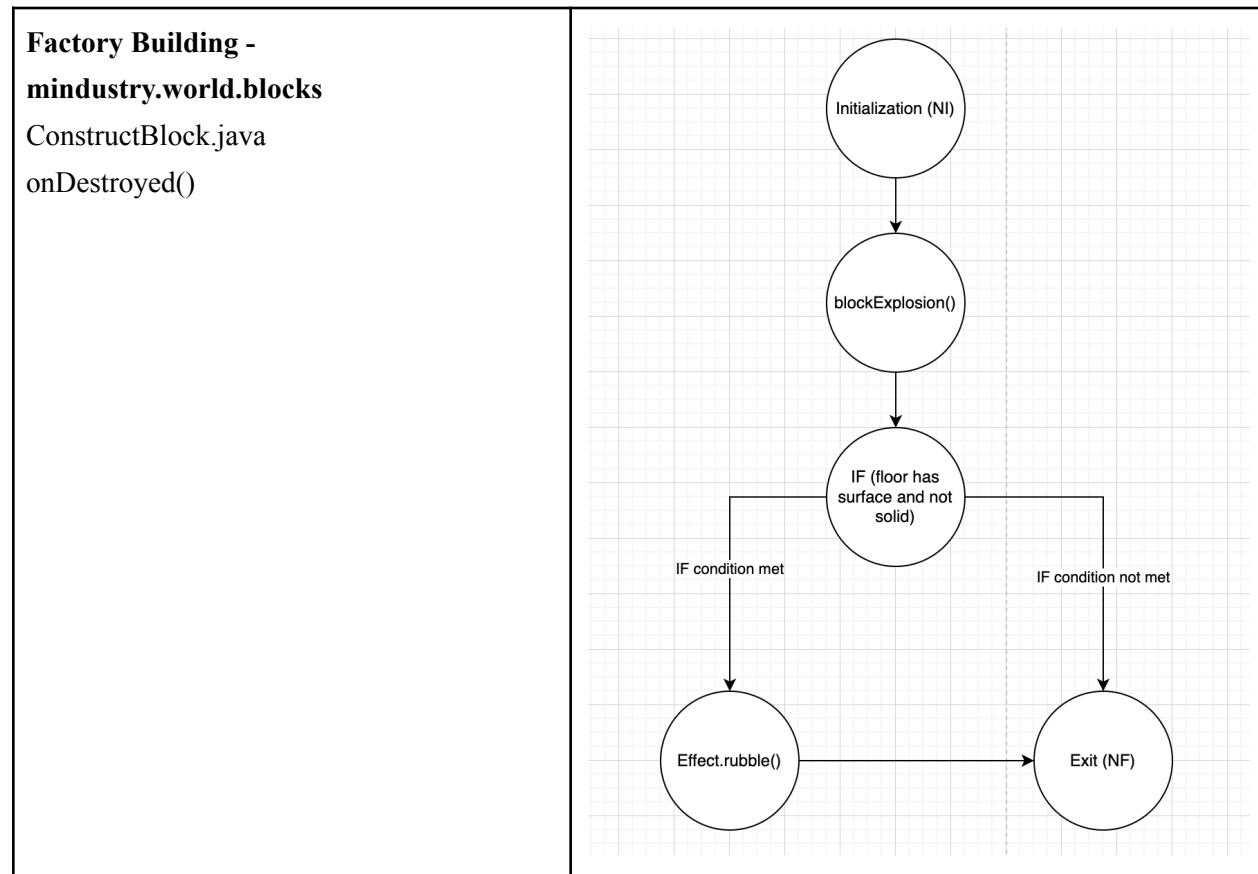
Test Case	Input Value	Expected Result	Actual Result	Pass/Fail
1 (Valid)	Valid Tile object with a block type of "copper-wall" at position (10, 10)	Block state is not changed, no resources are dropped, and no items are produced	As expected	Pass
2 (Invalid)	Valid Tile object with a block type of "copper-wall" at position (-1, -1)	IllegalArgumentException is thrown	IllegalArgumentException is thrown	Pass
3 (Lower boundary value)	Valid Tile object with a block type of "copper-wall" at position (0, 0) and a previous health of 50	Block state is not changed, no resources are dropped, and no items are produced	As expected	Pass
4 (Upper boundary value)	Valid Tile object with a block type of "thorium-reactor" at position (100, 100) and a previous health of 0	Block state is updated to air, resources (3 thorium) are dropped, and items (1 power) are produced	As expected	Pass

All test cases pass, which means that the updateTile() method handles all possible input values correctly, including edge cases and boundary values, based on the actual values used for testing.

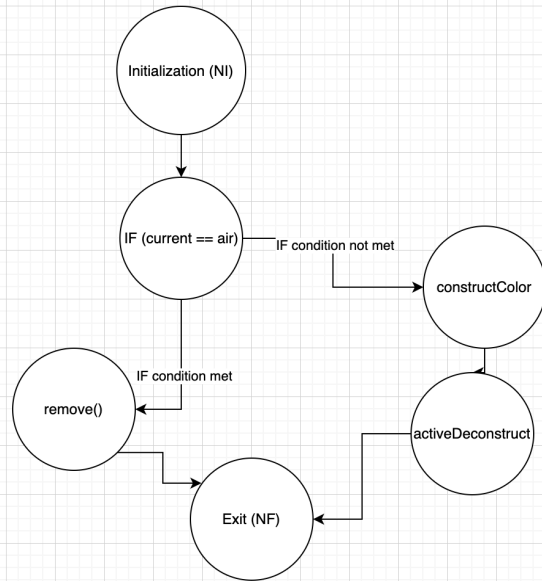
## Control Flow Graph (CFG):

You will need to draw CFGs and provide the corresponding analysis based on different coverage methods for control flow.

### Boundary Value Analysis for Factory Building - Mindustry.world.blocks (constructBlock.java)



**Factory Building -**  
**mindustry.world.blocks**  
ConstructBlock.java  
updateTile()



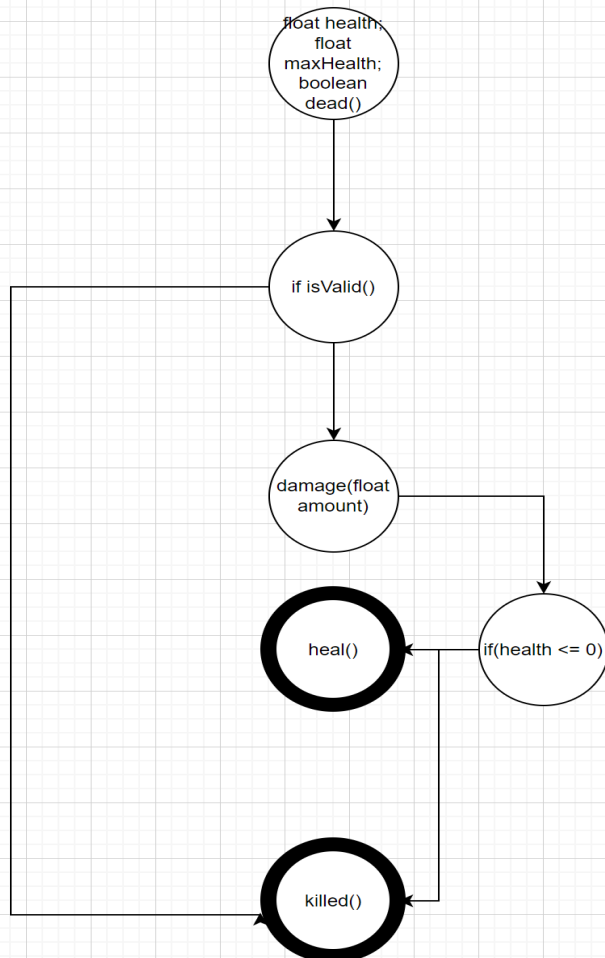


## Data Flow Graph (DFG)

### Data Flow Graph

Health and Damage -  
mindustry.entities.comp/  
HealthComp

Health.java



## Logic-based Testing

Feature #1 - Factory Building - Mindustry.world.blocks (constructBlock.java)

Logic-based testing for onDestroyed():

Test Case	Predicate	Clause	Expected Output
1	!drops.isEmpty()	true	Calls dropItems() method with drops argument
2	drops.isEmpty()	false	Does not call dropItems() method
3	group != null	true	Calls updateGroups() method
4	group == null	false	Does not call updateGroups() method

### #3: Health and Damage: Mindustry.entities.comp.health (Health.java)

Input Space Partitioning - Boundary Value Analysis (isValid())

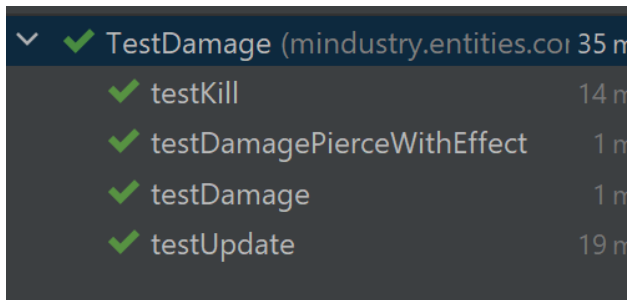
Test Case	Input Value	Expected Result	Actual Result	Pass/Fail
1 (Valid)	Player has health in the following value range; $100 \leq \text{health} < 0$	The player receives damage and its health reduces	As expected	Pass
2 (Invalid)	Player has health, $\text{health} = 0$	isValid() evaluates to false and the player is considered killed	As expected	Pass
3 (Lower boundary value)	Player health = 1	The player receives at least one damage and is killed	As expected	Pass
4 Upper boundary value	Player health = 100	The player receives damage and its health reduces	As expected	Pass

## Coverage

Overall, we achieved close to 100% class coverage and 50% line coverage in the classes that we tested.

Some sample test results are shown below:

### Feature #3 Health & Damage Testing

A screenshot of a test runner interface showing the results for the TestDamage class. The class name is 'TestDamage (mindustry.entities.combat)' with a green checkmark and a dropdown arrow. Below it, four test methods are listed, each with a green checkmark and a line count: 'testKill' (14 lines), 'testDamagePierceWithEffect' (1 line), 'testDamage' (1 line), and 'testUpdate' (19 lines).

✓	TestDamage (mindustry.entities.combat)	35 lines
✓	testKill	14 lines
✓	testDamagePierceWithEffect	1 line
✓	testDamage	1 line
✓	testUpdate	19 lines

The coverage below is for class, method, and line respectively:

A screenshot of a coverage summary bar for the HealthComp class. It shows the class name 'HealthComp' with a blue icon, followed by three coverage metrics: 100% (1/1) for class coverage, 43% (7/16) for method coverage, and 56% (17/30) for line coverage.

HealthComp	100% (1/1)	43% (7/16)	56% (17/30)
------------	------------	------------	-------------

## Fault types

Critical faults are those which interfere with the core functionality of the software. No critical or major faults were discovered in the software. A list of non-critical faults is listed below:

- Minor performance faults:
  - In some instances, the software appeared to be laggy. The lag is particularly pronounced in multiplayer games with more than 3 people.
  - When the map gets crowded with designs, players, enemies, and bullets, there appears to be some lag
- Enemy “confused” behavior
  - In some instances, enemy wave units appear disoriented in certain maps
- Incorrect Documentation
  - As the game is being build and updated on a near daily basis, the documentation is out of date in some places
- Lack of Existing Tests
  - There is minimal testing done by the contributors to this software, leading to very low coverage

## Potential difficulties and barriers

JUnit testing: The main challenge with JUnit testing was that it was challenging to test complex interactions between different units of code for Mindustry.

Boundary value testing: Difficulty with boundary value testing in the Mindustry game was determining the appropriate boundaries for various input parameters. Additionally, it was challenging to identify all the boundary conditions that need to be tested.

Control flow testing: Difficulty with control flow testing in the Mindustry game is that there may be many possible paths that the code can take, making it challenging to identify and test all possible scenarios.

Data flow testing: Data flow testing involves testing how data is used and manipulated within the code. One potential difficulty with data flow testing in the Mindustry game is that there may be complex interactions between different parts of the code, making it challenging to identify and test all possible data flow scenarios.

Logic-based testing: Difficulty with logic-based testing in the Mindustry game was that there were many complex logical interactions between different parts of the code, making it challenging to identify and test all possible scenarios.

Overall, testing a complex software application such as a game can be challenging and time-consuming. This is mainly due to the thorough understanding of the software's functionality, as well as the ability to identify and test all possible scenarios. Additionally, it requires the use of multiple testing techniques to ensure that the software is tested comprehensively.

## Appendix

### JUnit Test Class and Methods

<p><b>Tower Defense -</b> <b>mindustry.type.weapons</b></p> <p>BuildWeaponTest.java</p>	<pre>@Test public void testNoAttack() {     assertTrue(buildWeapon.noAttack); }  @Test public void testPredictTarget() {     assertFalse(buildWeapon.predictTarget);}  @Test public void testDisplay() {     assertFalse(buildWeapon.display);}  @Test public void testUpdate() {     // Create a dummy Unit and WeaponMount for testing     Unit unit = new Unit();     WeaponMount mount = new WeaponMount();     // Test that shoot is always false     buildWeapon.update(unit, mount);     assertFalse(mount.shoot);     assertTrue(mount.rotate);}</pre>
<p><b>Factory Building -</b> <b>mindustry.world.blocks</b></p> <p>AttributesTest.java</p>	<pre>@Test public void testGetAndSet() {     Attributes attributes = new Attributes();     attributes.set(Attribute.damage, 10);     assertEquals(10, attributes.get(Attribute.damage));}  @Test public void testClear() {     Attributes attributes = new Attributes();     attributes.set(Attribute.damage, 10);     attributes.set(Attribute.health, 20);</pre>

	<pre> attributes.clear(); assertEquals(0, attributes.get(Attribute.damage)); assertEquals(0, attributes.get(Attribute.health));}  @Test public void testAdd() {     Attributes attributes1 = new Attributes();     attributes1.set(Attribute.damage, 10);     attributes1.set(Attribute.health, 20);      Attributes attributes2 = new Attributes();     attributes2.set(Attribute.damage, 5);     attributes2.set(Attribute.health, 15);     attributes1.add(attributes2);     assertEquals(15, attributes1.get(Attribute.damage));     assertEquals(35, attributes1.get(Attribute.health));}  @Test public void testAddWithScale() {     Attributes attributes1 = new Attributes();     attributes1.set(Attribute.damage, 10);     attributes1.set(Attribute.health, 20);      Attributes attributes2 = new Attributes();     attributes2.set(Attribute.damage, 5);     attributes2.set(Attribute.health, 15);     attributes1.add(attributes2, 0.5f);     assertEquals(12.5f, attributes1.get(Attribute.damage));     assertEquals(27.5f, attributes1.get(Attribute.health));} </pre>
<p><b>Health &amp; Damage - mindustry.entities.comp</b></p> <p>HealthComp.java</p>	<pre> @Test public void testDamage() {     hc.maxHealth = 100f;     hc.health = 50f;     hc.damage(20f);     assertEquals(hc.health, 30f, 0.001f); }  @Test public void testUpdate() {     hc.hitTime = 0.5f;     hc.update(); } </pre>

	<pre>         assertTrue(hc.hitTime &lt; 0.5f);     }     @Test     public void testDamagePierceWithEffect() {         hc.maxHealth = 100f;         hc.health = 50f;         hc.damagePierce(20f, true);         assertTrue(hc.health &lt; 50f);     }     @Test     public void testKill() {         hc.health = 0.5f;         hc.kill();         assertTrue(hc.dead);         assertTrue(hc.health &lt;= 0);     } </pre>
<p><b>Tower Defense -</b>  <b>mindustry.entities.abilities</b></p> <p>ForceFieldAbility.java</p>	<pre>     @Test     public void testUpdateIncreasesShield() {         when(unit.shield()).thenReturn(100f);         forceFieldAbility.update(unit);         verify(unit).shield(100.1f);     }      @Test     public void testUpdateDecreasesAlpha() {         forceFieldAbility.alpha = 0.5f;         forceFieldAbility.update(unit);         assert(forceFieldAbility.alpha &lt; 0.5f);     }      @Test     public void testDrawWithShield() {         when(unit.team()).thenReturn(Pal.accent);         when(unit.shield()).thenReturn(100f);         forceFieldAbility.draw(unit);         verify(unit, times(2)).team();         verify(unit, times(2)).shield();     }      @Test </pre>



	<pre>public void testDisplayBars() {     Table mockTable = mock(Table.class);     when(unit.shield()).thenReturn(100f);     forceFieldAbility.displayBars(unit, mockTable);     verify(mockTable).add(any(Bar.class));     verify(mockTable).row(); }</pre>
--	---