Group 7 Phase 3 Report

Unit Tests:

For the unit testing, we wrote tests for each of the main classes in our project. Some of the key classes tested were for Enemy, Player, CollisionChecker, Sound, and KeyHandler.

Test Class	Key Test Cases	Feature
EnemyTest	testEnemyInitalization()	Ensures correct initialization for position, speed, direction
	testEnemyRestart()	Verifies enemy resets to default position and direction on restart
	testCollisionWithEnemy()	Checks collection detection with other enemies
CollisionCheckerTest	testCollisionWithPlayer(Right, Left, Up, Down)Movement()	Checks collision detection with player and barriers while player is moving in every possible direction
PlayerTest	testPlayerInitialization()	Verifies if the Player object initialized correctly.
	testPlayerRestart()	Checks that the restart method resets player state
	testPlayerImageLoading()	Ensures that the getPlayerImage method loads images without exceptions.
SuperObjectTest	testWorldCoordinatesSetterGetter()	Verifies that the world coordinates (X, Y) are set and retrieved accurately.
	testSolidAreaCoordinatesSetterGetter()	Validates setting and retrieving specific coordinates within the solid area.
	testNameSetterGetter()	Validates that the object's name can be set and retrieved correctly.
SoundTest	testPlay()	Checks that the play() method starts playing the sound clip without issues.
	testInitialization()	Ensures the sound URLs are correctly initialized and point to valid .wav files.
	testLoop()	Ensures the loop() method functions correctly without throwing exceptions.
KeyHandlerTest	testKeyReleased()	Validates that keys are correctly marked as released after being pressed
CollisionCheckerObjectTest	testNoCollision()	Validates that no collision is detected when the entity is positioned away from objects.

Integration Tests:

Some important interactions that we tested were:

Test Class	Key Test Case	Feature
EnemyTest	testUpdateEnemy()	Ensures enemy moves in player’s direction Ensures correct logic is applied when an enemy collision occurs
	testDetermineDirection()	Confirms direction updates correctly based on player position Tests all 4 possible directions
TileManagerTest	testLoadMap()	Checks if the map was loaded from the input file correctly
PlayerTest	testPlayerMovement()	Simulates player movement based on key presses, requiring interaction between Player, KeyHandler, and the game world.
	testExitDoorInsufficientClovers()	Verifies the game state when interacting with the exit door with insufficient clovers Involves interaction with GamePanel, the UI, and Player state.
	testNegativePoints()	Verifies the "you lost" state when points are negative Tests the interaction between Player, GamePanel, and music system.

	testPickClover()
	
	testPickCarrot()
	
	testPickMushroom()
		Tests the pickObject method in isolation by simulating a specific scenario
		Tests the interaction between Player, Objects, and music system.

KeyHandlerTest	testKeyPressed()
	testGuideStateKeyPress()
	testPauseToggle()
		Tests the interaction between KeyHandler and GamePanel by ensuring proper transitioning of game states.

MouseListenerTest	testGuideButtonClickInMenuState()
	
	testStartButtonClickInMenuState()
	
	testPLayAgainButtonClickInYouLostState(
	)
		Verifies that the mouse interaction in MouseListener correctly triggers a state change in GamePanel

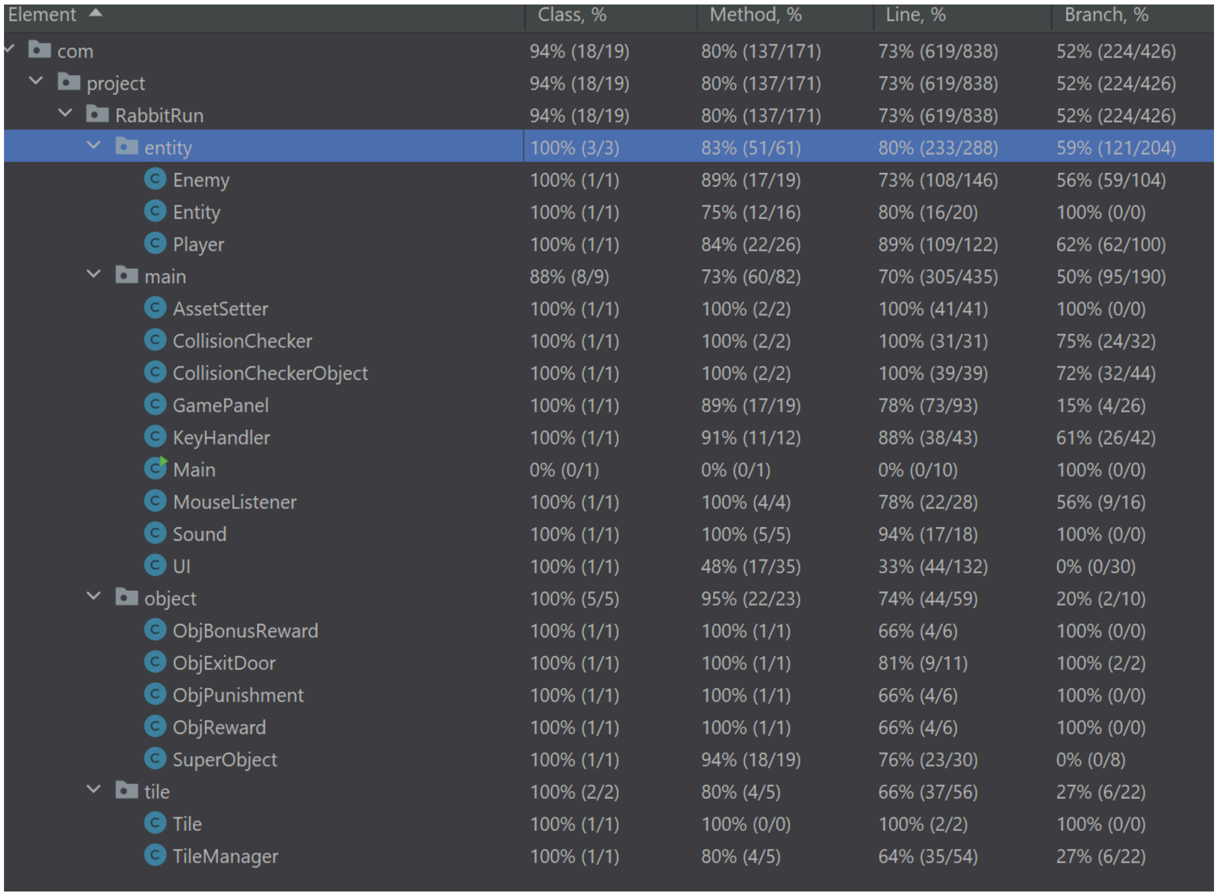
AssetSetterTest	testSetObjectPlacesCorrectTypes()	Ensures that the correct types of objects (e.g., rewards, punishments) are placed on the map.
	testSetObjectPlacesCorrectCoordinates()	Verifies that objects are placed at their intended coordinates on the game map.
	testObjectsAreNotNull()	Checks that all initialized objects in the game panel are not null after setup.

CollisionCheckerObjectTest	testCollisionFromRight()
	testCollisionFromLeft()
	testCollisionFromAbove()
	testCollisionFromBelow()
		Ensures collisions are correctly detected when the entity moves into an object from any direction.

	testMultipleObjects()	Ensures that collisions are accurately detected for multiple objects in the game panel.

Test Quality and Coverage:

To ensure quality tests were written, we prioritized having comprehensive coverage by trying to achieve high line and branch coverage where possible. We avoided having duplicate test cases by making sure each test focused on a unique feature or method. Furthermore, we designed the test cases to be independent of one another, preventing dependencies between tests, which made it easier to fix test cases. Lastly, assertions were used carefully to verify the expected test outcomes.



Element	Class, %	Method, %	Line, %	Branch, %
com	94% (18/19)	80% (137/171)	73% (619/838)	52% (224/426)
project	94% (18/19)	80% (137/171)	73% (619/838)	52% (224/426)
RabbitRun	94% (18/19)	80% (137/171)	73% (619/838)	52% (224/426)
entity	100% (3/3)	83% (51/61)	80% (233/288)	59% (121/204)
Enemy	100% (1/1)	89% (17/19)	73% (108/146)	56% (59/104)
Entity	100% (1/1)	75% (12/16)	80% (16/20)	100% (0/0)
Player	100% (1/1)	84% (22/26)	89% (109/122)	62% (62/100)
main	88% (8/9)	73% (60/82)	70% (305/435)	50% (95/190)
AssetSetter	100% (1/1)	100% (2/2)	100% (41/41)	100% (0/0)
CollisionChecker	100% (1/1)	100% (2/2)	100% (31/31)	75% (24/32)
CollisionCheckerObject	100% (1/1)	100% (2/2)	100% (39/39)	72% (32/44)
GamePanel	100% (1/1)	89% (17/19)	78% (73/93)	15% (4/26)
KeyHandler	100% (1/1)	91% (11/12)	88% (38/43)	61% (26/42)
Main	0% (0/1)	0% (0/1)	0% (0/10)	100% (0/0)
MouseListener	100% (1/1)	100% (4/4)	78% (22/28)	56% (9/16)
Sound	100% (1/1)	100% (5/5)	94% (17/18)	100% (0/0)
UI	100% (1/1)	48% (17/35)	33% (44/132)	0% (0/30)
object	100% (5/5)	95% (22/23)	74% (44/59)	20% (2/10)
ObjBonusReward	100% (1/1)	100% (1/1)	66% (4/6)	100% (0/0)
ObjExitDoor	100% (1/1)	100% (1/1)	81% (9/11)	100% (2/2)
ObjPunishment	100% (1/1)	100% (1/1)	66% (4/6)	100% (0/0)
ObjReward	100% (1/1)	100% (1/1)	66% (4/6)	100% (0/0)
SuperObject	100% (1/1)	94% (18/19)	76% (23/30)	0% (0/8)
tile	100% (2/2)	80% (4/5)	66% (37/56)	27% (6/22)
Tile	100% (1/1)	100% (0/0)	100% (2/2)	100% (0/0)
TileManager	100% (1/1)	80% (4/5)	64% (35/54)	27% (6/22)

General Observations:

- **Strong Line Coverage:** The majority of classes have high line coverage (above 70%), indicating that most of the functionality was exercised by the tests.
- **Decent Branch Coverage:** Some classes achieved moderate branch coverage (above 50%), but others had lower results due to untested branches in certain methods (e.g., TileManager).

Coverage Gaps and Explanations:

UI Class:

- **Reason for Low Coverage:** Testing graphical components like rendering and UI updates was deemed unnecessary since these involve visual output, which is difficult to automate without specialized tools.
- **Impact:** This lowered both line and branch coverage for the class.

Enemy and Player Classes:

- **Draw Methods Not Tested:** The draw methods in these classes were excluded from testing because they were graphics-related, similar to the UI class.
- **Impact:** This affected both line and branch coverage in these classes.

Main Class:

- **Reason for No Tests:** The Main class handles game initialization, including creating a JFrame and setting up the game environment. Since this is primarily GUI-related, it was excluded from testing.
- **Impact:** The class remains completely untested, contributing 0% to both line and branch coverage.

TileManager Class:

- **Low Branch Coverage (27%):** The draw functionality, which involves rendering tiles, was not tested for the same reasons as the UI and draw methods in other classes.
- **Impact:** Resulted in reduced branch coverage despite achieving decent method coverage (80%).

Getters and Setters:

- **Excluded from Tests:** These are simple, single-line methods that return or set values, and thus were not prioritized for testing.
- **Impact:** Affects line coverage slightly but does not impact branch coverage, as these methods typically lack complex logic.

Key Interactions and Integration Testing:

- **Map System:** The integration test for loading maps from an input file ensured that the system could correctly read and process the map data. This critical interaction was tested effectively, contributing to high coverage in related classes like GamePanel and TileManager.

Findings:

Achieving high test coverage was more challenging than expected, as it required careful planning to cover edge cases and complex logic. However, this helped reveal hidden bugs and unanticipated issues.

Some of our classes contained auxiliary methods that could be separated into their own classes. Originally, the collision checker class handled all collisions for objects as well as barriers. However, this resulted in extremely low branch coverage as collisions between the player and barriers had been tested in isolation, while the class still contained a method for checking collisions between the player and objects. Therefore, we opted to separate the collision checker into two classes. One class tests for collision between the player and barriers, while CollisionCheckerObject tests for collision between the player and placeable objects. This separation allowed us to test the methods in isolation and increased branch coverage to 75% for collision checker and 72% for collision checker object. From this we learned that classes that contain too many methods or that try to do too many things are difficult to test in isolation and do not provide accurate branch coverage results when developing unit tests.

Some changes made to our code was the addition of missing getters and setters for some variables. Also, the updateEnemy() method was refactored by turning a portion of the method’s logic into its own function. This improved the code's testability and readability.

During testing, we identified two critical bugs: a missing .wav extension in the Sound class path, which caused the game to not produce any sounds when a player collided with a reward object, and a logic issue in the Player class, where points and exit door conditions were only checked if a key was pressed. These issues were resolved during the testing phase, improving the functionality and robustness of the code. The process highlighted the importance of thorough testing to uncover hidden flaws and ensure better code quality.