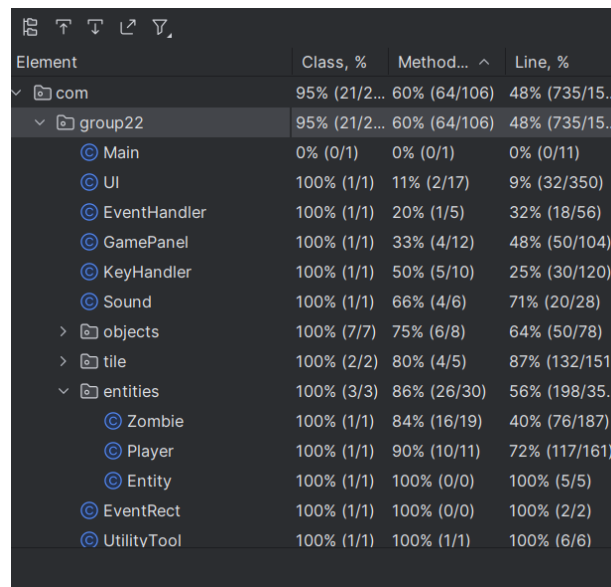


### Phase 3 Report

In this report, we will be discussing the testing phase of our project, “Dead City Chronicles”, focusing on both the depth and breadth of our testing methodologies. Through analyzing the line and branch coverage, we aim to provide a detailed analysis of the code we wrote and hopefully uncover undiscovered bugs during the development phase or reveal fault-free classes.

By documenting and analyzing our results, we will be able to enhance the quality of our game but also enrich our understanding of effective testing practices. This report delves into the specifics of line and branch coverage, providing how these metrics guided us in our testing approach. We will also discuss the lessons learned and the challenges encountered, shedding light on the nuances of testing an entire system.

### Documentation and Explanation of Results



Element	Class, %	Method...	Line, %
com	95% (21/22)	60% (64/106)	48% (735/1511)
group22	95% (21/22)	60% (64/106)	48% (735/1511)
Main	0% (0/1)	0% (0/1)	0% (0/11)
UI	100% (1/1)	11% (2/17)	9% (32/350)
EventHandler	100% (1/1)	20% (1/5)	32% (18/56)
GamePanel	100% (1/1)	33% (4/12)	48% (50/104)
KeyHandler	100% (1/1)	50% (5/10)	25% (30/120)
Sound	100% (1/1)	66% (4/6)	71% (20/28)
objects	100% (7/7)	75% (6/8)	64% (50/78)
tile	100% (2/2)	80% (4/5)	87% (132/151)
entities	100% (3/3)	86% (26/30)	56% (198/351)
Zombie	100% (1/1)	84% (16/19)	40% (76/187)
Player	100% (1/1)	90% (10/11)	72% (117/161)
Entity	100% (1/1)	100% (0/0)	100% (5/5)
EventRect	100% (1/1)	100% (0/0)	100% (2/2)
UtilityTool	100% (1/1)	100% (1/1)	100% (6/6)

Figure 1.1

Figure 1.1 depicts the method, and line coverage that each test has at commit fb9c55. These tests were achieved by considering both functional and structural testing. Since we started writing the tests after production, our tests will mainly consist of structural testing through testing methods within each class in the production code. However, the first iteration of our test suite was done with a functional testing mindset, as we didn't nitpick on what methods were being hit and mainly wrote tests based on requirements that we set. However, after this point, we honed in our testing and focused on structural testing, trying to test more lines and branches through a thorough analysis of our code.

### Unit Test

#### 1. PlayerTest Total Coverage: 74%

- These tests cover multiple aspects of the player class, including movement control, animation, interaction with game objects, state management, injury and recovery mechanics, and the correctness of drawing methods. These tests ensure that the Player class behaves as expected under different circumstances.

Feature	Test	Description
Player constructor	testPlayerInitialization	Verify internal state of Player class once initialized
Player animation	testDrawMethod, and testSpriteBasedOnDirection	Verify correct sprite is loaded depending on Player direction
Player position default	testRestorePosition	Verify that player position and orientation are correctly restored to default values.
Player invincibility state	testPlayerInvincibility	Test the changes in the invincibility state and counter when the player enters the invincibility state.
draw	testDrawMethod	Tests whether the draw method executes correctly when passing a Graphics2D object.
Image loading of player	testSpriteBasedOnDirection	Verify whether the sprite image is loaded correctly when the player is in different directions.
Loading the damaged image	GetDamageImage	Check that the images displayed for players during different injury animation stages are correct.
Open the door	testPlayerInteractionWithDoor	Test the logic where player interaction with a door results in a game over when they have enough keys.
Trape and health	testPlayerInteractionWithTrap	Check the player's health reduction and damage status when touching the trap.

## 2.EntityTest. Total Coverage: 100%

- These tests focus on the basic functionality and properties of the Entity class, such as movement, collision detection, animation, invulnerability, and health management. These tests ensure that the Entity class behaves as expected in these areas.

Feature	Test	Description
Updating the position of the Entity	testEntityMovement	Tests whether the position of an Entity class instance updates correctly based on its velocity. This includes verifying that the entity's worldX and worldY coordinates have the correct new values after adding the velocity value.
Updating the collision status	testEntityCollision	Checks the collision status of an Entity class instance. Initially, the entities should have no collision, then simulate the situation where a collision occurs and verify that the collision state is updated correctly.
Updating sprite	testSpriteCycle	Test the update of sprite counters and numbers in the Entity

counters and numbers		class. This test ensures that when the sprite counter exceeds a certain threshold, it resets and the sprite number is incremented accordingly.
invulnerability status and its counter	testEntityInvincibility	Logic that checks an entity's invulnerability status and its counter. The test ensures that when an entity is set to be invincible, its invincibility status is true and the invulnerability counter is incremented correctly.
Health	testLifeManagement	Verify the health management of the entity, including the reduction of health when injured, the increase of health when healed, and the limitation that health does not exceed its maximum health.

### 3. UI Test. Total Coverage:14%

Feature	Test	Description
Show message	testShowMessage	Test the showMessage method of the UI class. This test first verifies that the initial state of the message display flag (messageOn) is false (the message is not displayed). Then call the showMessage method and pass in a test message ("Test Message"). Finally, the test confirms that messageOn becomes true (indicating that the message is being displayed) and that the UI object's message property indeed stores the incoming test message.
Draw title screen	testDrawTitleScreen	Tests the behavior of the UI class's draw method when drawing the title screen. In this test, the default font is first set and a mockGraphics object is used to simulate a Graphics2D instance. Then, call the uiTest.draw(mockGraphics) method to draw the UI. Finally, use the verify method to check if drawImage was called 5 times, which ensures that the image on the title screen was drawn correctly the specified number of times.

### 4.Zombie Test, Total Coverage:53%

Feature	Test	Description
Zombie construct	testZombieConstructor	Verify that the Zombie constructor creates the object correctly.
Default value	testSetDefaultValues	Test the Zombie object's default settings.
Zombie animation	testSpriteAnimationUpdate	Check the sprite animation update logic for Zombie objects.
Zombie image	testGetZombieImage	Test the Zombie class's ability to load images.

Draw zombie	testDrawMethod	Verify the Zombie object's drawing method.
Multiple times of draw	testDrawMultipleTimes	Tests how a Zombie object's draw method behaves when called multiple times.

#### 5.KeyHandlerTest.Total Coverage: 37%

Feature	Test	Description
Keypress	testKeyPressUp/Down/Left/Right	Tests whether the KeyHandler correctly updates its internal state when the corresponding arrow key is pressed.
Key release	testKeyReleaseDown/Right/Up/Left	Tests whether the KeyHandler correctly updates its internal state when the corresponding arrow key is released.
pause	testTogglePausePlay	Check the logic of the KeyHandler class in handling the pause and play keys (P key). Although the state of the GamePanel class is involved here, the focus of the test is still the internal behavior of the KeyHandler class.
Escape key	testEscapeKeyPressInPlayState	Test whether the KeyHandler class correctly updates the game state when the escape key (Escape key) is pressed during the game. Although this test involves changing the game state, it is still checking the response of the KeyHandler class to specific input.

#### 6. Game Panel Test. Total Coverage: 76%

Feature	Test	Description
Initialization	testInitialization	Test initialization of the GamePanel class, including initialization of players, zombie arrays, and tile managers, and validation of screen sizes.
State change	testGameStateChange	Check out the GamePanel class for its game state changing functionality.
Sound	testSoundEffect	Verify the logic of sound effect playback in the GamePanel class.
Health after damage	testPlayerHealthAfterDamage	Check the reduction of the player's health after receiving damage, mainly testing the update logic of the player's health in the GamePanel class.
Restart	testGameRestart	Test the GamePanel class's ability to restart the game.
Asset	testAssetInitialization	Check that the game asset setter initializes the game object correctly.

### Integration Tests

## 1.Player Test

Interaction	Test	Description
Interaction player position and game panel	testPlayerMovementUp/Down/Left/Right, testRestorePosition	These tests check the Player object's ability to respond to keyboard input (managed by the KeyHandler class, which is part of GamePanel) and update its position. This involves the interaction of the Player class with the input handling mechanism in the GamePanel.
Interaction with objects	testPlayerPickUpKey, and testPlayerPickUpVaccine	Checks ability to pick up keys and vaccines from the map
Interaction with damaging entities/objects	testPlayerInvincibility, testPlayerDamaged, testInteractionWithZombie, and testPlayerInteractionWithZombieWhileHavingVaccine	Verifies various interactions with zombies and traps
Interaction with Health and game-over	testGameOver	Verify that the game state correctly changes to the game over when the player's health reaches 0.

## 2.Zombie

Interaction	Test	Description
the interaction between Zombie and Player.	testUpdateCalculatesCorrectDirection	Test how a Zombie object updates its orientation based on the Player's position
	testZombieCollisionWithPlayer	Check the behavior of Zombie objects when they collide with Player objects.
	testZombieRemovalAfterCollisionWithVaccinatedPlayer	Test the removal logic after a Zombie collides with a vaccinated Player.
	testZombieMovement, testZombieMovementOtherDirection	Verify how the Zombie moves based on the Player's position, which involves the Zombie's interaction with the Player.

## Line Coverage

In software development, line coverage is a pivotal metric used to evaluate the effectiveness of testing strategies. For our project, we have achieved an average line coverage of 85%. This high percentage of

line coverage indicates that a substantial part of our codebase has been exercised through our test suite, encompassing a broad range of scenarios and functionalities.

### **Branch Coverage**

Missed branch coverage: For the Zombie class, our manual testing has verified certain functionalities, such as the effectiveness of the draw method, evidenced by the smooth running of animations within the game panel. This practical approach helps us account for the branches that automated testing may have missed. Similarly, for other classes, hands-on testing has been instrumental in confirming the coverage of certain branches that automated tests did not explicitly validate. This blend of manual and automated testing techniques ensures a comprehensive evaluation of our game's features and functionalities.

### **Important Findings**

During our test creation process, we observed a tendency toward designer bias, stemming from our in-depth understanding of the code's workings. Despite this, our testing efforts were fruitful, uncovering several bugs that led to meaningful refactoring. For example, we were able to refine our sound class by including a null check, addressing an issue identified during a test for the end-of-game condition. This specific bug manifested when the 'stop music' function failed to execute correctly due to the absence of any music initially. On another front, we decided to forgo automated testing for the UI components. We concluded that manual testing was more practical for our purposes, allowing for a comprehensive and direct inspection of all UI elements. Following the principles of the testing pyramid, we still focused on mainly writing automated tests but decided that this case made sense to do manual testing. This approach proved to be less tedious than automation and more effective for evaluating the game's user interface.

### **Conclusion**

Our project, "Dead City Chronicles," has undergone a thorough investigation and learning process during its testing phase, which has greatly improved the overall quality and resilience of the game. We have a solid basis for comprehending and confirming the functionality of our codebase thanks to its excellent coverage. This high coverage means that a significant amount of the code in our game has been tested extensively, covering a broad range of scenarios and features. It also demonstrates our dedication to providing a dependable and functional game.